Simulation modélisation traitement des données - Une approche pragmatique

D. Buskulic

Simulation, modélisation, traitement des données Une approche pragmatique

D. Buskulic

LAPP / Université de Savoie, B.P. 110
74960 Annecy-le-Vieux

RESUME

La simulation, la modélisation et le traitement des données en physique corpusculaire est un sujet extrêmement vaste, que nous aborderons de manière pratique en posant le problème particulier posé à la physique des hautes énergies, suivi de quelques notions de programmation orientée objet. Des outils existants de simulation (GEANT4) et d'analyse (ROOT) seront présentés pour donner un contenu concret au cours.

ABSTRACT

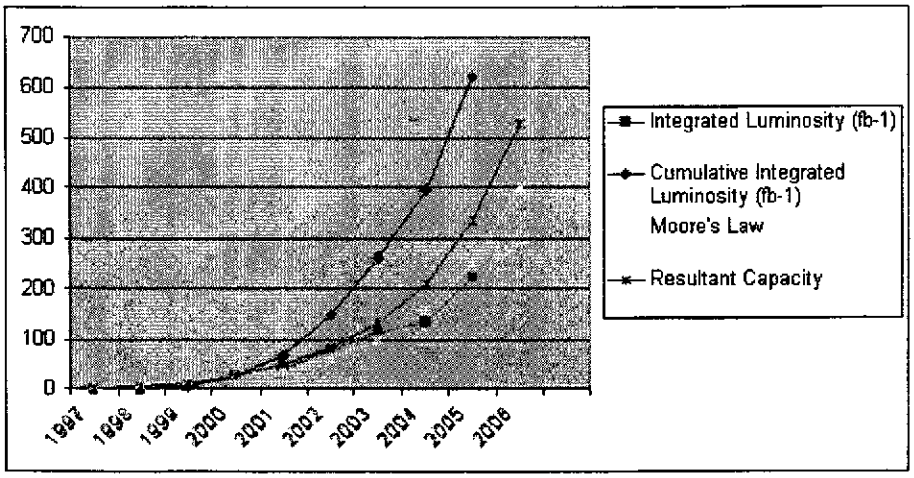
Simulation, modeling and data analysis in particle and nuclear physics is an extremly vast subject. We will approach it in a practical way by evaluating the particular problem that arises in high energy physics, followed by a few notions of object oriented programming. Existing simulation (GEANT4) and analysis (ROOT) tools will be presented to give a practical content to this course.

I. GENERALITES

I.1 Contraintes sur les outils de Physique Corpusculaire

Avant de concevoir des outils de simulation ou de traitement des données, il est bien évidemment nécessaire de connaître les spécificités liées aux données en question.
Le premier paramètre qui saute aux yeux est la quantité de données à traiter dans nos expériences. Accumuler un téraoctet (10^{12} octets) est aujourd'hui chose courante. Les expériences de physique des particules sur collisionneur (LHC) ou bien les expériences de collisions d'ions lourds (RHIC, LHC) vont, dans les années qui viennent, accumuler jusqu'à quelques pétaoctets (10^{15} octets) par an. Pour donner un ordre de grandeur, ceci correspond à 20000 bandes actuelles, ou 100 000 DVD-RAM double face ou encore 1 500 000 exemplaires de l'Encyclopaedia Universalis !
De plus, dans ce flot de données, seulement 1 événement sur 10^6 est intéressant, c'est-à-dire correspond à autre chose que du bruit de fond. Et seulement 1 sur 10^9 REELLEMENT intéressant, correspondant à autre chose que ce qui est déjà connu et étudié.
Tout ceci nécessite le développement d'outils de "Data Mining", c'est à dire de recherche et d'isolement d'événements très rares dans une très grande quantité de données. La recherche d'une aiguille dans une botte de foin, en quelque sorte.
Mais tous les problèmes ne vont-ils pas être complètement effacés par le développement rapide de l'informatique et des capacités de stockage ? On estime que l'évolution des puissances de calcul et des capacités des moyens de stockage doublent tous les 18 mois. Ce constat est connu

sous le nom de "loi de Moore" du nom de l'un des cofondateurs de la société Intel. Le graphique suivant montre qu'il ne faut pas trop compter sur cette évolution, bien qu'elle soit exponentielle.



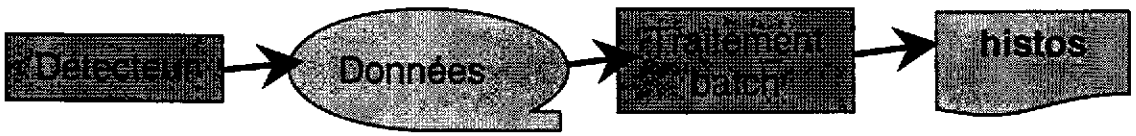
En effet, la quantité de données prises par l'expérience Babar au cours de son fonctionnement croît plus rapidement que les capacités disques prévues par la loi de Moore.

Les conséquences de cette très grande quantité de données accumulées sont multiples. Le plus visible est l'importance que vont prendre les outils de gestion de données, comme les logiciels de gestion de bases de données, mais également les outils d'analyse statistique. De plus, l'analyse réalisée par les physiciens requiert un accès efficace aux données et aux très gros programmes développés. Qui dit grande quantité de données dit également très grande quantité de calculs nécessaires au traitement de ces données, comme la simulation, la reconstruction et l'analyse. La puissance de calcul estimée varie entre 1000 et 10000 gigaflops (1 flop = 1 opération sur un réel par seconde) soit la puissance de 1000 et 10000 PC actuels regroupés en fermes de calcul. On voit ici que le calcul sera parallèle et, dans le futur,probablement distribué sur plusieurs centres répartis géographiquement dans des pays différents.

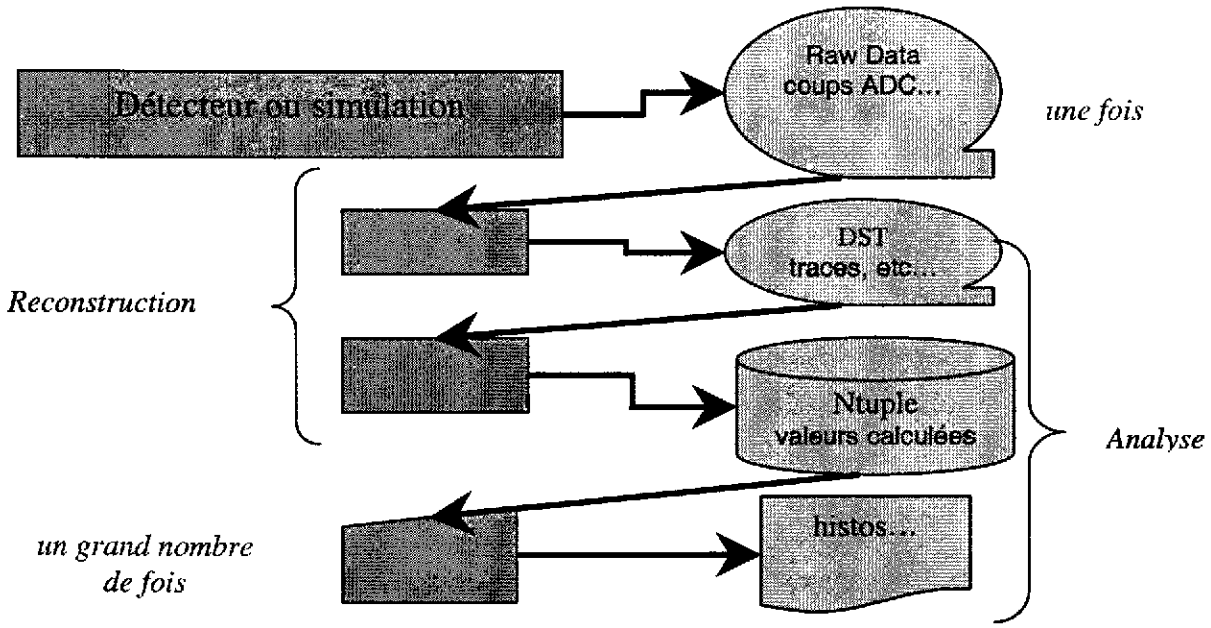
I.2 Critères et fonctionnalités à respecter

I.2.1 Historique de l'analyse

Pour définir les critères et fonctionnalités que devront présenter les logiciels futurs, il est intéressant de détailler la façon dont se passait une analyse de physique des hautes énergies jusqu'à aujourd'hui.



Un détecteur fournit des données qui sont enregistrées sur bande. L'utilisateur soumet un travail en batch (hors ligne) à une ferme de calcul distante. Ce travail consiste la plupart du temps a générer des résultats sous forme d'histogrammes ou de données très réduites (ntuples) qui seront analysées localement.
Un modèle plus complet est le suivant :



Comme ci-dessus, on part de données brutes (Raw Data) contenant l'amplitude des diverses réponses des détecteurs, par exemple le nombre de coups des divers convertisseurs numériques-analogiques. Ces données sont prétraitées pour digérer les informations. On obtient des DST (Data Summary Tapes) contenant par exemple les positions des traces ou bien les impulsions des particules reconstruites. Un pas supplémentaire est franchi dans le traitement lorsqu'on utilise les données DST pour extraire les informations physiques comme le type de particule ou bien l'existence d'une particule parent de celles détectées. Une dernière phase du traitement consiste à extraire les informations physiques sous forme d'histogrammes, d'incertitudes sur les résultats, etc...

Mais ce schéma, bien qu'utilisé avec succès jusqu'à présent ne suffira pas à traiter l'énorme quantité de données qui sortira du LHC.

I.2.2 Tâches à effectuer

Quelles sont donc les tâches à effectuer ? nous en avons déjà citées quelques-unes :

- Calibrer les signaux
- Extraire les composants élémentaires (traces)
- Reconnaître les particules individuelles
- Reconnaître les caractéristiques physiques de l'événement
- Extraire les quantités physiques

Ceci, dix millions de fois! Reste à

- Etudier la physique
- Mesurer, publier...

Nous devons également considérer une grande variété de situations. Aussi bien près du détecteur, par exemple les algorithmes de reconstruction, les études sur l'évolution des performances, que hors-ligne :

- Etude d'une grande quantité d'événements
- Etude d'un seul événement marquant

- Analyse de signal
- Comparer théorie et résultats expérimentaux

I.2.3 Problèmes cruciaux liés aux tâches à effectuer

Et parmi toutes ces tâches, quels sont les problèmes cruciaux rencontrés ?
Le premier problème est la localisation des données. Plusieurs péta octets doivent être distribués, ce qui implique plusieurs centres de stockage interconnectés.
Par ailleurs, le problème du traitement est primordial. Où les données vont-elles être traitées ? On ne peut sérieusement envisager qu'un traitement également distribué, au plus près des données. Qui imaginerait de transporter des péta octets de données pour les analyser ? Peut-être, un jour, avec des réseaux beaucoup plus performants...
Des problèmes d'une telle dimension ne peuvent être résolus qu'internationalement. Un exemple en est l'expérience BaBar qui a deux sites de stockage et calcul, Stanford près de San Francisco et Lyon (CCIN2P3)

I.2.4 Influence sur les modèles d'environnement de travail

Les problèmes ci-dessus exposés vont influencer les environnements de travail et outils que l'utilisateur physicien va devoir manipuler. Ils devront concilier deux points de vue parfois divergents :
Le point de vue de l'utilisateur qui a besoin de "toucher et voir", de manipuler les données pour acquérir une compréhension des problèmes. La boucle expérimentale se retrouve alors également dans l'analyse. De plus, le physicien doit pouvoir traiter aussi bien de petites quantités de données "à fond" que de grosses quantités dans des traitements par lot (batch).
Le point de vue du concepteur du système est légèrement différent. Il pense en termes de fermes de production centralisées, de calcul local ou distribué, de modèles informatiques (Software Engineering). La séparation Frameworks vs Composants, les méthodes de conception prennent de l'importance.
Comparons donc les modèles d'environnements de travail que manipulent les uns et les autres.
Les physiciens sont habitués à la boîte à outils. Celle-ci doit avoir des outils puissants, qui permettent de contrôler tout le processus depuis le début. Mais cette approche n'est pas adaptée aux flux de données actuels.
Une approche qui permet de se concentrer davantage sur le problème est l'approche composants ("plug-ins") où l'on assemble le système final à l'aide de briques logicielles élémentaires. On peut alors oublier certains détails lors de l'analyse, tout en courant le risque que ces détails n'en soient pas... La question en suspens est peut-on faire tout ce que l'on veut avec cette approche ?
Une approche "services" est peut-être un bon compromis, où l'utilisateur envoie ses données subir un certain traitement auprès d'acteurs spécialisés (calcul, visualisation...). Des exemples des approches actuelles seront données dans la suite de ce cours.

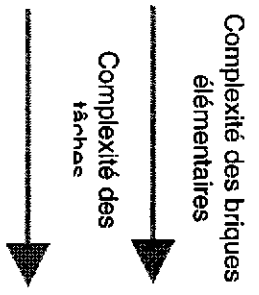
II. PROGRAMMATION ET LANGAGES ORIENTES OBJET

La notion d'"orienté objet" est aujourd'hui très courante. Il est important d'en montrer le sens et la finalité, les outils logiciels que les physiciens ont à manipuler étant de plus en plus basés sur ces notions. Nous ne chercherons pas à faire un cours complet sur la programmation orientée objet mais plutôt à en faire ressortir les notions importantes et à montrer ce qui en fait l'utilité.

II.1 Motivations

La motivation de base de la notion d'orienté objet tient en une phrase : "La quantité et la complexité des données rendent nécessaire une certaine forme d'abstraction des détails de base". Pour appuyer notre propos, considérons l'évolution de la fabrication des ordinateurs de 1950 à nos jours.

1950	Le composant de base est le tube électronique
1960	Les transistors apparaissent
1970	On passe aux circuits intégrés
1980	Le microprocesseur naît
1990	Les microcontrôleurs intègrent plusieurs éléments dans un seul boîtier
2000	On sait réaliser un ordinateur sur une puce



Ce qui est remarquable est la montée en complexité des ordinateurs, en même temps que leur miniaturisation. Un ingénieur des années 50 assemblait des tubes pour réaliser les premiers ordinateurs. Celui des années 70 mettait sur un circuit imprimé des circuits intégrés, réalisant chacun ce que réalisaient plusieurs centaines de tubes. Aujourd'hui, l'unité de base est le processeur, capable de s'interfacer à l'extérieur. Bref, on assemble toujours des briques, mais elles sont devenues de plus en plus complexes.
Quel concepteur de PC s'occupe de savoir ce que fait chacun de 20 millions de transistors dans un microprocesseur ?

L'une des observations réalisées lors du développement de logiciels pour les grandes expériences est qu'il y a un couplage dans l'ensemble du code. Au début, la conception des algorithmes se fait à l'aide d'outils d'analyse, lors de la réalisation et mise au point du détecteur. Certaines parties de ce code d'analyse vont remonter le diagramme du paragraphe I.2.1 pour passer de l'analyse vers les programmes de reconstruction, puis peut-être vers les déclenchements en ligne. Ceci vient du désir légitime des concepteurs de ne pas réécrire le code déjà développé.
Mais est-il possible, sans généraliser et adapter le code, de lui faire faire des tâches légèrement différentes ? Que doit-on faire pour réellement avoir à réécrire le moins de code possible ?

II.2 Un bon logiciel c'est...

Avant de se lancer dans la suite, pouvons-nous définir ce qu'est un "bon" logiciel ?
Le maître mot pourrait alors être : "Vous écrivez vos logiciels pour les autres !"
Un bon logiciel doit (devrait ?) :
Avoir une structure aisément compréhensible

- Pouvoir être facilement débogué
- Autoriser facilement le changement d'une partie sans affecter les autres parties
- Avoir un code modulaire et réutilisable
- Etre simple à maintenir et mettre à jour
- Etc...

La programmation orientée objet (POO) veut se rapprocher de cet idéal, elle vous aidera à écrire de bons logiciels... mais...

Ce n'est pas la panacée ! L'utilisation d'un langage dit "orienté objet" (en abrégé OO) ne garantit pas une "programmation orientée objet". D'ailleurs, un code mal écrit en C++ est pire qu'un code

mal écrit en Fortran (l'auteur en sait quelque chose...). Les langages OO ne sont destinés qu'à simplifier une bonne programmation OO
Le langage doit être considéré comme un outil pour arriver à l'orientation objet.

II.3 Programmation orientée objet : histoire

La naissance de la programmation orientée objet remonte à plus de 30 ans. 30 ans de développements, d'expérience et de pratique de la programmation. Le premier langage que l'on peut qualifier de OO est Simula67, créé en 1967 dans un laboratoire pour réaliser des simulations. Sa portée a été faible à cause de la spécificité de son champ d'application. En 1980 est né Smalltalk80, premier langage "universel" orienté objet. Ont suivi Lisp, Clu, Actor, Eiffel, Objective C. C++ et Java sont apparus plus récemment.
Qu'est-ce qui a poussé le développement de ces langages ? La motivation principale a été la crise du logiciel entre les années 1980 et 1990. Il s'agissait d'une crise de croissance de la complexité. Le matériel devenait de plus en plus puissant et de moins en moins cher. Le logiciel, par contre, devenait de plus en plus complexe et cher à développer. Est alors naturellement arrivé le besoin de rendre le code réutilisable.
L'idée de base qui a émergé est de cacher les détails de l'implémentation d'un algorithme, programme ou structure au monde extérieur, de ne montrer qu'une interface.

II.4 Principe et concepts de base

L'idée, qui vient d'être esquissée, est de représenter le comportement du monde réel sous une forme qui cache les détails de l'implémentation. Chaque "élément" ou objet va cacher son comportement interne. On n'aura plus à se soucier que des interactions avec l'extérieur. Lorsque ca fonctionne, la POO permet de penser en termes de domaine élargi du problème, d'assembler des briques, quelle que soit leur complexité interne.
Trois groupes d'idées fondamentales caractérisent la Programmation Orientée Objet :

- La classe / l'objet et la notion associée d'encapsulation
- Les hiérarchies de classes et l'héritage
- L'abstraction et le polymorphisme

II.4.1 Classes, Objets et Encapsulation

Classes et objets

La POO est une manière de programmer centrée sur les objets (Quoi ? Ce qui fait) plutôt que sur les procédures (Comment ? Ce qui est fait). Un objet contient des données internes et une description des actions qu'il peut effectuer, de son comportement. À ce titre, les objets et leur comportement sont très fortement liés. Un objet ne peut exécuter une action prévue pour un autre objet !
On définit la notion de classe comme un type d'objet. Un objet particulier sera appelé une instance de la classe. Une classe représente un type de "chose" dans le système, un objet représente une chose particulière.
Par exemple, une classe "trace" est un type, la trace particulière "π^+ n° 23" est un objet.
Finalement, la POO voit la programmation comme une activité de simulation de comportement. Ce qui est simulé, ce sont des objets représentés par une abstraction dans le programme. Ce n'est pas tout à fait un hasard si le premier langage orienté objet était un langage destiné à la simulation...
Pour illustrer le fait que les classes sont responsables de leur comportement, on peut donner un exemple de définition de classe écrit en C++ :

```
class Impulsion
{
public :
    Impulsion(double x0, double y0, double z0, double e);
    ~Impulsion();
    Impulsion&   operator= (const Impulsion &
    Impulsion&   operator+ (const Impulsion &
    double       Module();
}
```

Définition d'une classe décrivant une impulsion

Définition d'un comportement (calcul du module dans ce cas)

Dans cet exemple, on définit une classe d'impulsion, avec ses comportements. L'écriture du code source correspondant sera faite dans l'implémentation, dans une autre partie du fichier ou dans un autre fichier.
On peut noter en passant qu'en C++ (mais pas en Java), on peut définir les opérateurs standard (=, +, -, *, /) pour une classe. Ceci permet par exemple de définir la notion d'"addition" pour une classe, lorsqu'elle a un sens. Un exemple est donné ci-dessus pour la classe impulsion. On peut alors écrire dans un programme

```
Impulsion a,b,c;
c=a+b;
```

Ceci peut améliorer la lisibilité du code mais nous n'en conseillons pas l'usage à moins de bien savoir ce que l'on fait.
Nous n'avons presque pas parlé des données internes d'une classe. Ces données internes sont appelées données membres.

class TGTextLine

```
private:

protected:

public:
      TGTextLine TGTextLine()
      TGTextLine TGTextLine(TGTextLine* line)
      TGTextLine TGTextLine(const char* string)
      TGTextLine TGTextLine(TGTextLine&)
    virtual void ~TGTextLine()
  static TClass* Class()
         void Clear()
         void DelChar(ULong_t pos)
         void DelText(ULong_t pos, ULong_t length)
         char GetChar(ULong_t pos)
      ULong_t GetLineLength()
        char* GetText(ULong_t pos, ULong_t length)
         void InsChar(ULong_t pos, char character)
         void InsText(ULong_t pos, const char* text)
virtual TClass* IsA() const
    virtual void ShowMembers(TMemberInspector& insp, char* parent)
    virtual void Streamer(TBuffer& b)
         void StreamerNVirtual(TBuffer& b)
```

Data Members

```
private:

protected:

      char* fString  Line of text
    ULong_t fLength  Lenght of line
TGTextLine* fPrev    previous line
TGTextLine* fNext    next line

public:
```

Les données membres d'une classe ont une relation d'appartenance (possède un). Par exemple, une classe décrivant une particule possède une donnée décrivant la charge.
Dans l'exemple ci-dessous, une classe décrivant une ligne de texte possède une donnée membre qui est un entier décrivant la longueur de la ligne.

Encapsulation

En plus des données internes et de la description des fonctions et comportements que doit avoir un objet décrit par une classe, il faut que le programmeur décrive par un code source ces comportements, c'est à dire qu'il implémente les fonctions spécifiques à la classe. Cette

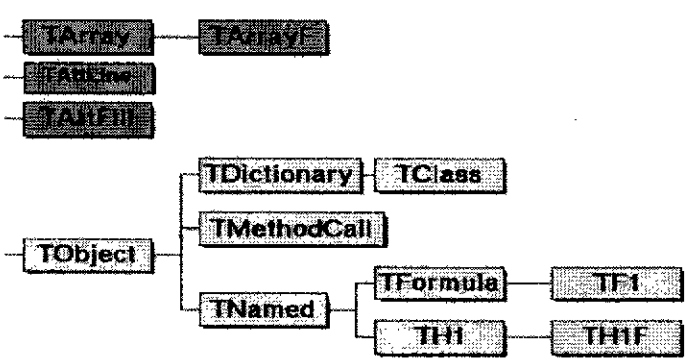
implémentation peut être très complexe, et nous avons vu que la crise du logiciel des années 1980-1990 venait de la complexité croissante des programmes. Tout était relié à tout et les logiciels devenaient impossibles à maintenir et à faire évoluer. Pour mettre un peu d'ordre, la POO exige que les détails de l'implémentation, le code source décrivant les comportements, ainsi que les données internes d'une classe, soient invisibles de l'extérieur. De l'extérieur, on ne voit qu'une interface qui permet d'accéder aux fonctions et données. Par extérieur, on entend les codes et objets qui utilisent la classe.
Ceci s'appelle "l'encapsulation".
Les données membres de la classe doivent être privées, accessibles seulement à travers des fonctions membres (qu'on appelle méthodes) du type Set/Get.
Une conséquence de ceci est que les changements de l'implémentation interne ne doivent pas modifier la façon dont on accède et utilise la classe extérieurement. Le programmeur peut décider de modifier le code interne d'une classe, pour l'améliorer, sans que les objets extérieurs s'en rendent compte, ou qu'il soit nécessaire de les modifier eux-mêmes.
Un exemple de déclaration de classe est donné ci-dessous (en C++) :

	class Impulsion	class Impulsion
	{	{
	...	...
	private:	private:
Données internes (privéees)	double m_Px; double m_Py; double m_Pz; double m_E;	double m_P; double m_Theta; double m_Phi; double m_E;
	public:	public:
Méthodes publiques	double GetP() const; void SetP(double p);	double GetP() const; void SetP(double p);

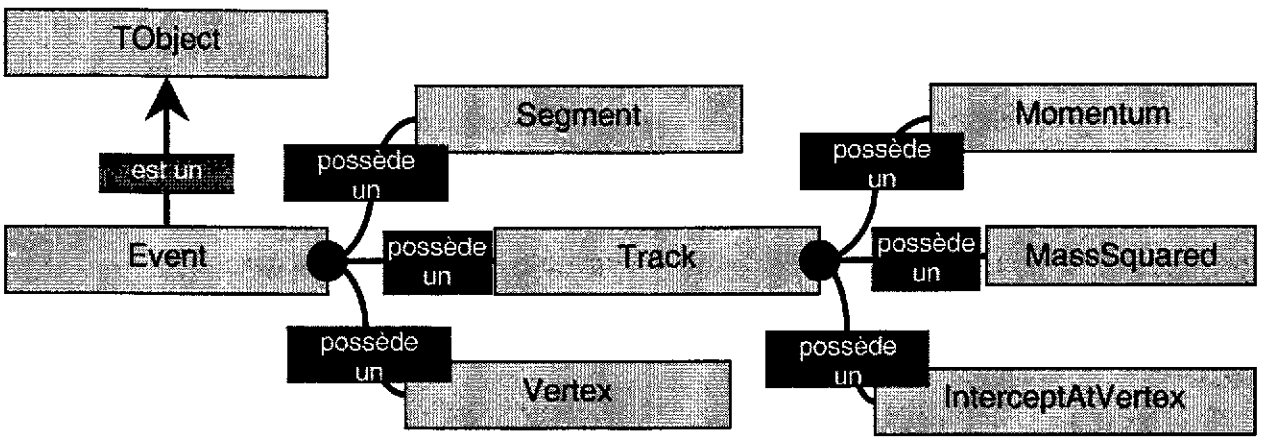
Peu importe comment le vecteur impulsion est codé de façon interne, en coordonnées cartésiennes ou sphériques, l'extérieur ne voit que les fonctions qui permettent d'accéder aux informations.

II.4.2 Hiérarchies de classes et héritage

La deuxième notion importante en POO est l'héritage. C'est un moyen de dériver une nouvelle classe, appelée classe dérivée, de classes préexistantes, appelées classes de base. Par exemple si une classe décrit l'ensemble des véhicules roulants (classe de base), on décrira les camions à l'aide d'une classe dérivée, les automobiles à l'aide d'une autre, les tracteurs à l'aide d'une troisième.
La nouvelle classe dérivée peut utiliser le code de sa ou ses classes de base. A travers l'héritage, on peut créer une hiérarchie de classes qui partagent du code et des interfaces. L'héritage est une méthode pour gérer la complexité.
Un exemple de hiérarchie est le suivant :



L'héritage correspond à une notion d'identité (est un...). Si vous ne savez pas comment organiser vos classes, posez vous la question "est-ce que c'est un...". Si la réponse est oui, vous pouvez utiliser l'héritage. Par exemple une voiture "est un" véhicule roulant. La classe voiture peut donc dériver de la classe véhicule roulant. Un histogramme à deux dimensions (classe TH2) est un histogramme (classe TH1), donc la classe TH2 peut dériver de la classe TH1. Etc...
Ne pas confondre la notion d'identité et la notion d'appartenance. Le schéma suivant montre la différence pour une classe "Event":



Un événement (classe Event) EST UN objet (classe TObject) qui POSSEDE une ou des traces (classe Track) qui elle même POSSEDE une quantité de mouvement (classe Momentum) .

II.4.3 Abstraction et polymorphisme

Abstraction

On peut définir des classes destinées uniquement à donner naissance à d'autres classes par héritage. Ce sont des classes "abstraites". Ceci permet

- De définir une interface générale ("abstraite") à une notion ou à un ensemble d'objets sans avoir à écrire de code.
- De simplifier la modularité et la portabilité du code. Par exemple GEANT4 (logiciel dont nous reparlerons) est libre de tout choix de techniques d'entrées/sorties ou d'histogrammation. De même, le GUI (Graphical User Interface) et la visualisation sont complètement isolées du noyau de GEANT4 par l'intermédiaire d'interfaces abstraites. On peut ainsi "accrocher" le module de visualisation que l'on veut, ou celui adapté à la machine ou au système d'exploitation sur lequel on travaille.

Polymorphisme

Le polymorphisme (plusieurs notions/éléments/classes prennent la même forme) se décline en POO selon plusieurs aspects:

Polymorphisme : surcharge de fonctions et d'opérateurs

En POO, une fonction peut avoir le même nom, mais des arguments différents. Par exemple

```
Draw(TLine* ligne)
Draw(TBox* carre)
```

Bien que l'on ne trace pas de la même manière une ligne et un carré, on fait la même action, tracer. La distinction est faite par le compilateur sur le type des arguments ("signature" de la fonction). On parle de surcharge de fonctions.
Nous avons déjà vu qu'en C++, on peut également surcharger des opérateurs. Par exemple, la somme de deux histogrammes peut avoir un sens. On peut surcharger les opérateurs "+" et "=" pour permettre :

```
h3 = h1 + h2;
```

Nous vous avons déjà averti, ne le faites que si vous savez où vous mettez les pieds !

Polymorphisme : redéfinition de méthode

Lorsqu'une classe dérivée a besoin de réaliser une opération définie dans la classe de base mais un peu différemment, elle peut redéfinir la fonction membre de la classe de base. Par exemple la fonction "Draw". Supposons que la classe "TArrow" décrivant une flèche dérive de la classe "TLine" décrivant une ligne. Pour tracer une flèche, il faudra ajouter le code de tracé de la pointe de la flèche au code de tracé de la ligne. Ceci peut se faire dans la classe dérivée en redéfinissant la fonction membre "Draw". On aura une fonction Draw de TArrow et une fonction Draw de TLine. Il n'est pas interdit à TArrow d'utiliser le code de tracé de ligne contenu dans Draw de TLine.
Ce genre de redéfinition de méthode recèle des pièges redoutables, aussi vous conseillons-nous de vous reporter à un guide plus complet selon le language que vous adoptez.

Polymorphisme : Patrons de classes

C++ a la notion de polymorphisme paramétrique. Kézaco ?
Un patron de classe (template) permet de définir une classe indépendamment d'un type
Par exemple un tableau de nombres, qu'ils soient entiers, longs, float ou doubles (réels), est toujours un tableau et il serait inutile de réécrire du code juste en changeant le type de tableau.
Ceci est principalement utilisé dans la librairie STL (Standard Template Library).
Bien que les templates facilitent le développement, il le font un peu trop ! Le code peut rapidement devenir incompréhensible entre des mains non expertes. Un conseil personnel de l'auteur : en tant qu'utilisateur, n'allez pas au delà de l'utilisation de STL...

II.5 C++ et Java

II.5.1 C++

Le C++ a été à l'origine dans les laboratoires américains d'AT&T. L'architecte principal en a été Bjarne Stroustrup.
Ce langage a été conçu comme une extension du C qui prend certaines libertés avec les concepts POO "purs". Les principales "libertés" ainsi prises sont :

- L'encapsulation n'est pas obligatoire. On peut, si l'on veut, permettre aux codes extérieurs de voir et modifier les données internes d'une classe.
- On mélange les fonctions (procédures) du C et les classes et leurs fonctions membres. Conséquence : on peut très bien écrire un programme procédural en C++, sans aucune classe !

Pourquoi les auteurs de ce langage ont-ils été ammenés à réaliser ces choix ? Pourquoi pas de la POO pure ? La principale raison est que les programmeurs de l'époque étaient habitués au C. Le C++, bien qu'imparfait du point de vue de la POO, a permis une transition "acceptable" de la programmation procédurale vers la POO. Son principal défaut est que l'utilisateur a une trop grande liberté d'action. Il est facile de mettre le fouillis dans du code en mélangeant hardiment deux types de programmation très différents. Quelqu'un a déjà eu des démêlés avec des pointeurs ?

II.5.2 Java

L'origine de Java remonte à 1991, dans les cerveaux de la société américaine (encore !) Sun Microsystems.
Les objectifs de ses concepteurs étaient les suivants :
On voulait un langage simple, il a donc été choisi une syntaxe de type "C".
On voulait aussi un langage sûr : pas de manipulation de pointeurs, une vérification du code à l'exécution.
On voulait également un langage Orienté Objet : ni variables, ni fonctions globales, autrement dit pas de possibilité de faire de la programmation procédurale. Le programmeur est obligé de passer par des classes.
On le voulait robuste : ramasse-miettes, fortement typé, gestion des exceptions (ça, c'est pour les spécialistes...)
Enfin, on le voulait Indépendant de l'architecture sous sa forme "interprétée", nous y revenons un peu plus loin.
Bref, c'est de la POO "pure".

Java : langage orienté objet

En Java, TOUT est classe (pas de fonctions) sauf les types primitifs (int, float,...) et les tableaux. Toutes les classes héritent d'une classe de base java.lang.Object. Nous verrons que cette notion de classe de base générale a été reprise dans ROOT, logiciel décrit extensivement plus loin.
Une limitation a été introduite volontairement. Il est impossible de définir un héritage multiple. Une classe ne peut dériver de deux classes différentes à la fois, seul l'héritage simple est autorisé. Par exemple, une classe décrivant les voitures rouges ne peut pas à la fois dériver de la classe des objets roulants et de la classe des objets de couleur rouge. Ceci est possible en C++. Bien que ca puisse apparaître comme une limitation, cela évite pas mal de problèmes, que nous n'avons pas la place de décrire ici.

Java : avantages

Si l'on parle des avantages de Java, le premier qui vient à l'esprit est la portabilité. Tout programme Java peut s'exécuter sur n'importe quelle machine équipée d'une "machine Java". Lorsqu'on écrit un programme, le compilateur java génère du "byte-code", c'est à dire un code binaire indépendant du processeur. La machine Java va interpréter ce code binaire indépendamment de la plate-forme. Ces "Java Virtual Machine" sont présentes sur de nombreuses plateformes : Unix, Win32, Mac, Netscape, IE,...
Notons que la taille des types primitifs (entiers codés sur 4 octets par exemple) est indépendante de la plateforme, ce qui n'est pas, pour des raisons historiques, le cas en C++.
De plus, Java est toujours accompagné d'une librairie standard qui comprend des primitives graphiques ou de visualisation.
La robustesse est le deuxième élément clé. Elle est due principalement aux contraintes imposées au langage, beaucoup plus restrictif que le C++.

Java : différences avec le C++

Si l'on énumère "à la Prévert" les différences existant entre Java et C++, on obtient pour Java :

- Pas de structures, d'unions
- Pas de types énumérés
- Pas de typedef
- Pas de préprocesseur
- Pas de variables, de fonctions en dehors des classes
- Pas d'héritage multiple
- Pas de surcharge d'opérateurs

- Pas de passage par copie pour les objets
- Pas de pointeurs, manipulation de référence seulement

Java : l'avenir ?

Pour autant, avons nous la un langage parfaitement adapté à nos besoins ? Le coté "interprété" est sans doute le plus gros handicap. Il limite la vitesse de calcul et l'on sait combien cette vitesse est importante, au vu de la quantité de données à traiter. Peu de compilateurs natifs sont disponibles mais il n'est pas inenvisageable que cela change, par exemple avec le développement de compilateurs JIT (Just In Time)
Les avantages de robustesse et l'approche "OO pur" reste cependant très séduisants.
Une question qui reste en suspens est que, les utilisateurs, en physique corpusculaire commençant juste à intégrer les concepts OO et à basculer en partie vers le C++, peut-on leur demander encore un effort ? Et Quand ? Le fait que la syntaxe soit proche du C/C++devrait faciliter la transition, si transition il y a.
Et d'ici à ce que d'autres langages émergent...

III. LES OUTILS DE SIMULATION ET D'ANALYSE DISPONIBLES ET A VENIR

III.1 Outils de simulation

Les outils de simulation développés dans le passé pour les besoins de la physique corpusculaire se divisent en deux grands groupes. Premièrement la simulation de détecteurs, ou si l'on veut la simulation du passage des particules dans la matière, et deuxièmement les générateurs d'événement, simulant la production des particules et les collisions entre particules ou noyaux à des niveaux fondamentaux.
En ce qui concerne la simulation de détecteurs, pratiquement tous les efforts de la communauté des physiciens concernés en Europe se sont concentrés sur un outil : GEANT 3. Les générateurs d'événements étaient et sont beaucoup plus nombreux, citons par exemple Pythia, Venus, FLUKA,.... Ils évoluent au fil des progrès théoriques.
Pour le futur, le projet GEANT 4 est le nouveau projet dominant de simulation de détecteurs. Les générateurs d'événements anciens en FORTRAN pour la plupart, sont peu à peu soit interfacés aux environnements de travail nouvellement développés ou bien réécrits en C++. Ils ne cessent bien sûr pas d'évoluer quand à leur base théorique.

III.2 Environnements d'analyse

En ce qui concerne les environnements et outils d'analyse, ils ont par le passé été dominés par PAW, écrit au CERN il y a déjà une quinzaine d'années. Les grandes expériences avaient leur environnement de travail propre, avec souvent une utilisation de PAW pour l'analyse des données digérées.
Le futur se dessine mais de façon encore floue. Les candidats en lice sont nombreux et nous allons donner un petit aperçu de la diversité :

III.2.1 JAS (Java Analysis Studio)

C'est un environnement entièrement basé sur Java. Il en hérite les avantages et inconvénients. Les fonctionnalités inclues sont en gros celles dont nous avons besoin en Physique corpusculaire. JAS est très modulaire, indépendant du format de données. Il reste du travail à faire, mais le concept est intéressant.
Adresse web : http://jas.freehep.org/

III.2.2 Anaphe

Anaphe est le projet précédemment dénommé LHC++. Il est principalement basé sur des produits commerciaux.
Adresse web : http://anaphe.web.cern.ch/anaphe/

III.2.3 Open Scientist

Open Scientist est un environnement qui se veut extrêmement modulaire, reprenant l'approche "plug-in" poussée à son maximum.
Il essaye de récupérer et intégrer des codes libres existants.
Adresse web : http://www.lal.in2p3.fr/OpenScientist/

III.2.4 ROOT

On va en reparler de façon extensive...

III.2.5 Alors, lequel ?

On a en apparence le choix... mais les projets qui sont présentés ci-dessus ne peuvent pas encore prétendre à la maturité. Aussi présenterons-nous dans la suite celui qui est actuellement le plus développé et le plus utilisé dans le monde, ROOT. C'est un choix, l'auteur ne prétend pas à la neutralité. Mais nous invitons le lecteur à fouiner sur les sites web des autres projets.

IV. EXEMPLE D'UN OUTIL DE SIMULATION : GEANT 4

IV.1 Introduction

GEANT est un outil de simulation pour les détecteurs. Plus exactement, il réalise la simulation de l'interaction particule-matière depuis la production des particules dans les collisions jusqu'à la numérisation du signal dans l'électronique du détecteur. Il fournit une infrastructure générale pour :

- La description des géométries et matériaux
- Le transport de particules et leur interaction avec la matière
- La description de la réponse du détecteur
- La visualisation des géométries, des traces et coups

L'utilisateur s'occupe de développer le code pour le générateur d'événements primaire, la description de la géométrie du détecteur et la numérisation de la réponse du détecteur.
Historiquement, le programme appelé GEANT3 a été utilisé par la plupart des expériences de physique corpusculaire. Son développement est terminé depuis Mars 1994 (Geant3.21). Il a également été utilisé en médecine, dans la recherche spatiale,...
La diversité et la complexité des domaines d'application a donné système complexe, rendant sa maintenance difficile. La nécessité d'un outil flexible s'est faite sentir très fortement.
Bien qu'ayant rendu des services inestimables, GEANT3 est un exemple parfait de système qui a grossi au delà de ce que ses concepteurs pensaient en faire. Une mini-crise du logiciel en quelque sorte. Il devenait impossible d'ajouter une fonctionnalité ou de rechercher un bug, à cause de la structure trop complexe dominée par des contraintes historiques. Cela a conduit à repartir d'une page blanche en essayant de ne pas retomber dans le même schéma de développement.
GEANT4 est né de ces contraintes. Il s'agit d'une collaboration internationale qui a adopté des méthodes d'ingénierie logicielle (OOA pour Object Oriented Analysis et OOD pour Object Oriented Design). Le choix de l'orientation objet et C++ a été fait dès le début du projet.

Le début du projet, justement, remonte à Décembre 1994. La première version de développement (Geant4 0.1) date de Juillet 1999. Les versions ultérieures se sont succédées beaucoup plus rapidement. La dernière en date est Geant4 3.2 datant de Juin 2001. Les concepteurs de Geant4 estiment qu'il y a encore au moins 10 ans de maintenance et d'améliorations à venir.
Quels sont les avantages de Geant4 par rapport à Geant3 ?
Un point sur lequel peu de contestation est possible est celui de la flexibilité. Geant4 permet l'implémentation de modèles alternatifs, c'est-à-dire que l'on peut tester plusieurs types d'interactions en même temps. Il est de plus ouvert à une évolution future grâce à ses interfaces pour des logiciels externes. Et il est extensible, l'implémentation de nouveaux modèles et algorithmes se fait sans interférer avec le logiciel existant.
Enfin, tout est ouvert à l'utilisateur. Le choix des processus physiques et des modèles, le choix de l'interface graphique et de la technologie de visualisation.

IV.2 Simulation de détecteur dans un cadre OO

Que veut dire simuler un détecteur dans un cadre Orienté Objet ? Ou plutôt l'orientation objet a-t-elle un sens pour la simulation d'un détecteur ? En physique corpusculaire, la simulation revient à faire de la réalité virtuelle. Pour aider à la conception des détecteurs durant la phase de recherche et développement, pour comprendre la réponse du détecteur pour les études de physique, il y a une nécessité de modéliser les interactions particules-matière, la géométrie et les matériaux pour propager les particules élémentaires à l'intérieur du détecteur. Il faut également décrire la sensibilité du détecteur pour générer les données brutes (RAW data). Rappelons nous que les premières approches OO ont été faites dans un cadre de simulation, justement pour simuler la réalité. Simulation et Orienté Objet vont très naturellement de pair.
GEANT4 est une boîte à outils orientée objet fournissant les fonctionnalités nécessaires pour les simulations de détecteurs. Les bénéfices inhérents aux concepts OO, que nous avons détaillé dans les chapitres précédents, permettent au logiciel d'être facile à maintenir et à développer, modulaire et aisé à comprendre pour les non initiés (sur ce dernier point, nous attendons les réactions des non initiés...)

IV.3 Concepts de base

Détaillons les quelques concepts de base de Geant4 utiles au physicien.

IV.3.1 Le Run

Un run est un ensemble d'événements partageant les mêmes conditions de détection. Un "run" dans Geant4, comme dans une expérience réelle, commence par "BeamOn".
Dans un Run, l'utilisateur ne peut pas changer la géométrie du détecteur ou les réglages des processus de physique. Le détecteur n'est pas accessible durant un Run !

IV.3.2 L'événement (Event)

Au démarrage d'un traitement, un événement contient des particules primaires, issues du générateur d'événement physique.
Ces particules primaires sont poussées dans une pile (stack) puis traitées à tour de rôle. Le traitement d'une particule peut à son tour générer des particules qui seront poussées dans la pile. Lorsque la pile devient vide, le traitement de l'événement est terminé.
La classe G4Event représente un événement. On accède aux objets suivants à la fin du traitement :

- La liste des vertex primaires et des particules
- L'ensemble des trajectoires (en option)
- L'ensemble des coups dans les détecteurs (Hits)

- L'ensemble des digits (voir plus loin...)

IV.3.3 La trace (Track)

Une trace N'EST PAS l'ensemble des états d'une particule au cours du temps. Dans le jargon Geant4, il s'agit plutôt de l'instantané de l'état de la particule. A un instant donné, la trace décrit la position, le vecteur vitesse, etc... Dans le même ordre d'idées, un pas (Step) est une information de variation pour la trace (voir paragraphe suivant). D'après ce qui précède, une trace n'est pas un ensemble de pas. On peut presque dire qu'un pas est la "dérivée", au sens mathématique, d'une trace, bien que ce terme soit un peu galvaudé ici.
Une trace évolue au cours de son histoire dans le détecteur et disparaît lorsqu'elle a fini d'être traitée par le système. Ceci se produit si :

- Elle passe au delà du volume "monde", autrement dit au delà des limites du détecteur. Ces limites peuvent être très éloignées de ce que l'on entend d'habitude par détecteur. Pensez par exemple à l'atmosphère terrestre, qui sert parfois de détecteur. Raison pour laquelle on préfère définir un volume monde qui englobe tous les éléments utiles d'un détecteur.
- Elle disparaît par exemple lors d'une désintégration.
- Elle ralentit jusqu'à une énergie cinétique nulle et il ne reste plus de processus au repos à traiter.
- L'utilisateur décide de la tuer.

IV.3.4 Le pas (Step)

Un pas, comme nous venons de le voir, est une information de variation pour une trace. Il possède deux points dans l'espace et une information de variation (delta) pour la particule (variation d'énergie lors du pas, temps de vol, etc...)
Lors de la description du détecteur, l'utilisateur définit des volumes de matière (ou de vide...) que les particules vont traverser. Chaque point extrémité du pas connaît les caractéristiques des volumes définis dans le détecteur. Dans le cas où un pas conduirait une trace à sortir d'un volume, le point de fin se tient sur le bord et appartient au volume suivant.

IV.3.5 La trajectoire

Nous avons vu qu'une trace n'est pas l'ensemble des états d'une particule au cours du temps. Pourtant, il faut bien qu'un objet décrive cet historique. C'est le rôle dévolu à la "trajectoire".
Une trajectoire est l'historique d'une trace, ou plus exactement le résumé géométrique des états successifs d'une trace. Elle contient une information sur chaque pas fait par le trace. La classe définissant un point de trajectoire est G4TrajectoryPoint, celle définissant la trajectoire est G4Trajectory.
Il est possible de ne pas construire de trajectoire pour une trace. Mieux vaut ne pas mettre les traces des particules secondaires d'une gerbe dans des trajectoires. La mémoire occupée exploserait et dans le cas d'événements complexes, pourrait saturer alors que l'on n'a pas forcément besoin de l'historique des secondaires.
L'utilisateur peut créer sa propre classe de trajectoire dérivant des classes de base G4VTrajectory et G4VTrajectoryPoint. Ce faisant, on peut enregistrer n'importe quelle information additionnelle pour la simulation

IV.3.6 Processus physiques

Les processus physiques traités par Geant4 sont de trois types de base :

- Les processus au repos (ex : désintégration au repos)
- Les processus continus (ex: ionisation)

- Les processus discrets (ex: désintégration en vol)

Le transport, c'est-à-dire l'évolution de l'état des traces, est aussi considéré comme un processus. Ceci permet par exemple de traiter l'interaction avec les limites de volumes de la même manière logique que les autres processus.

IV.3.7 Détecteur sensible et coups (hits)

Lorsque l'on décrit les volumes d'un détecteur, certains pourront donner une information physique exploitable par l'extérieur. D'autres, bien que contenant de la matière et interagissant avec les particules, ne le pourront pas. Par exemple, un volume contenant du scintillateur produira de la lumière détectable alors qu'une plaque de plomb ne verra pas de détecteur sensible aux processus internes à cette plaque. Que fait-on lorsqu'un volume donne une information exploitable ? On lui associe un détecteur sensible. Dans l'exemple du scintillateur, ce sera un photomultiplicateur suivi de sa chaîne d'acquisition. Dans Geant4, chaque volume logique peut posséder un pointeur sur un détecteur sensible.
Un détecteur sensible, dans le jargon Geant4, produit des "coups" (hits). Un coup est un instantané de l'interaction physique d'une trace, ou une accumulation d'interactions d'un ensemble de traces, dans une région sensible de votre détecteur. Un détecteur sensible crée des coups (hits) en utilisant l'information donnée dans un objet G4Step. L'utilisateur doit fournir sa propre implémentation de la réponse du détecteur. Dans notre exemple de scintillateur, on peut ainsi se contenter de compter le nombre de photons sortant ou bien simuler toute la chaîne d'acquisition, y compris la numérisation. Les coups, appartenant à l'objet de la classe utilisateur, sont collectés et transmis dans un objet G4Event à la fin de l'événement

IV.3.8 Classes "Manager"

Pour faire tourner un programme tel que Geant4, il faut gérer la boucle d'événements, la visualisation, etc... Ceci est réalisé par ce que l'on appelle les classes "Manager". Nous ne donnerons que le nom des plus importantes, laissant au lecteur le soin de se reporter aux manuels utilisateur Geant4 qu'il trouvera aisément sur la toile.
Les classes de gestion d'un run, de l'interface avec le système de visualisation,... :

- G4RunManager
- G4SDManager
- G4UIManager
- G4FieldManager
- G4VVisManager
- ...

IV.3.9 En pratique

En pratique, que fait l'utilisateur pour faire tourner une simulation ?
En premier lieu, il doit écrire le programme "main". Rappelons que, bien que Geant4 effectue la gestion de la boucle d'événements, du transport des particules, etc... il ne s'agit pas d'un programme mais plutôt d'une bibliothèque de classes. Ceci laisse une liberté totale à l'utilisateur. La contrepartie est que celui-ci doit écrire un programme minimum qui appelle la boucle d'événements. Libre à lui d'écrire un programme plus complet ou complexe.
Dans le programme main, l'utilisateur doit construire un objet de la classe G4RunManager de gestion du run ou d'une classe dérivée. Il faut alors indiquer à cet objet les classes utilisateur décrivant respectivement la construction du détecteur, la liste des particules pouvant être générées, le générateur d'événement primaire. De façon optionnelle, on peut définir le manager de visualisation, la session GUI (Graphical User Interface) et des classes d'action utilisateur.
Toutes les classes dont nous venons de parler peuvent être les classes par défaut de Geant4, ce qui réduit le programme principal à quelques lignes mais ne fait strictement rien, ou bien des classes fournies par l'utilisateur. Dans ce cas, celui-ci part des classes virtuelles

- G4VUserDetectorConstruction
- G4VUserPhysicsList
- G4VuserPrimaryGeneratorAction

pour dériver ses propres classes selon le schéma et les méthodes imposées par les classes virtuelles. Il est possible de définir des actions utilisateur exécutées à chaque pas.
Nous venons en quelques lignes de décrire ce que l'utilisateur a à faire mais rien ne vaut de mettre les mains dans le cambouis. Pour les lecteurs intéressés, les exemples fournis avec la librairie Geant4 et le manuel utilisateur sont nombreux et vont du plus simple au plus compliqué. Bon courage !

IV.4 Caractéristiques

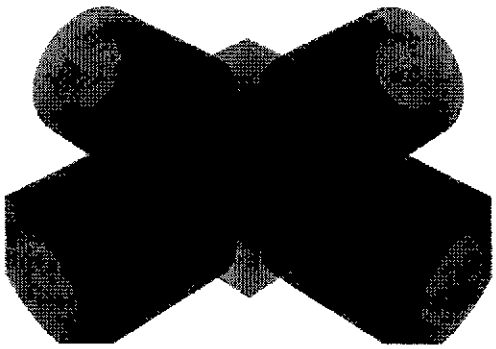
Voyons maintenant les caractéristiques de la librairie Geant4. Nous ne prétendrons pas à l'exhaustivité mais essaierons de donner une idée de sa richesse.

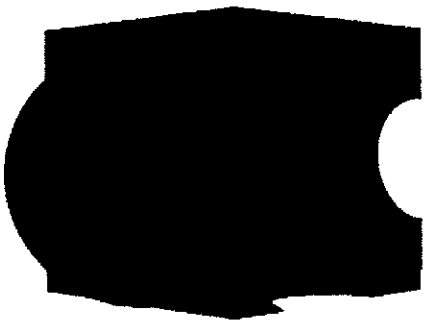
IV.4.1 Le noyau

Comme dans un système d'exploitation, la librairie est articulée autour d'un noyau qui gère la boucle d'événements. Le run manager peut différer le traitement d'une trace d'un événement sur le suivant, ce qui permet de gérer le "pile-up", l'accumulation de traces provenant de plusieurs interactions, soit parce que le détecteur particulier est trop lent vis à vis de la fréquence des interactions élémentaires, soit parce que plusieurs interactions par collision se produisent.
Le "Tracking", c'est-à-dire l'évolution des traces, est découplé de la physique, tous les processus sont gérés à travers la même interface abstraite. Il est indépendant du type de particule, ce qui donne la possibilité d'ajouter de nouveaux processus physiques sans le gêner.

IV.4.2 La géométrie

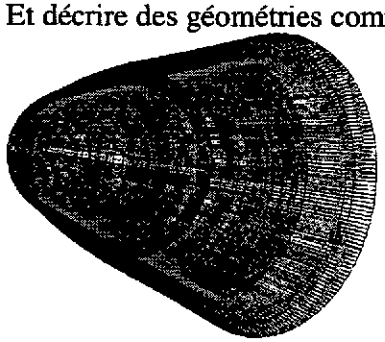
Pour décrire le détecteur et naviguer à travers lui, des outils géométriques sont présents. Une image valant mille mots, voici ce qu'il est possible de faire :



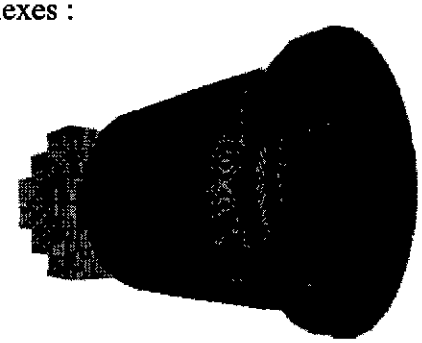


On peut faire des opérations booléennes sur les solides

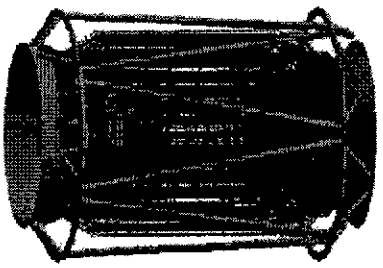
Et décrire des géométries complexes :



CMS



XMM - Newton



BaBar

IV.4.3 Physique

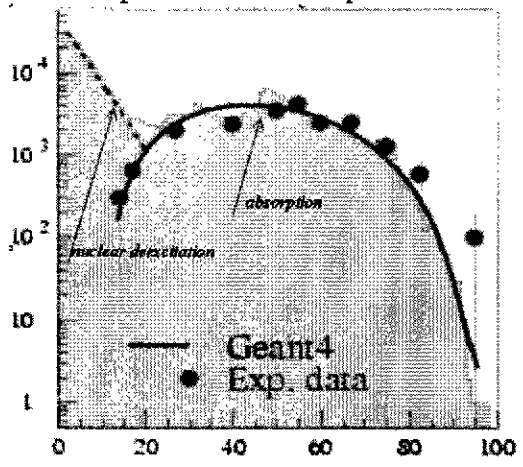
L'approche choisie pour la partie physique a été celle de la souplesse. Une grande variété de modèles physiques indépendants et alternatifs (possibilité d'avoir plusieurs modèles pour le même processus) ont été développés. Ceci revient à dire qu'il y a une distinction entre processus et modèle. Plusieurs modèles peuvent être utilisés pour le même processus.
Par ailleurs, il n'y a pas de "packages" ou de boites noires, les utilisateurs sont en prise directe avec la physique. Ceci permet une validation plus aisée des résultats expérimentaux.
La façon de traiter la physique électromagnétique et la physique hadronique sont les mêmes. Il y a utilisation de bases de données expérimentales du monde entier et une validation des résultats physiques au vu des données expérimentales.

IV.4.4 Physique électromagnétique

Les objets traités en physique électromagnétique sont les électrons et positons, les photons (γ, X et optiques), les muons, les hadrons chargés et les ions. Par rapport à Geant3, une extension vers les hautes énergies, telles que celles traitées au LHC et dans le cas de rayons cosmiques de haute énergie, est incluse.
Une extension vers les basses énergies existe également, pour les applications spatiales, médicales, expériences ν, spectroscopie d'antimatière, etc...
Les processus gérés sont très nombreux. Citons : la diffusion multiple, le bremsstrahlung, l'ionisation, l'annihilation, l'effet photoélectrique, la diffusion Compton, l'effet Rayleigh, la conversion γ, la production de paires e+e-, le rayonnement synchrotron, le rayonnement de transition, l'effet Cerenkov, la réfraction, la réflexion, l'absorption, la scintillation, la fluorescence, l'effet Auger (implémentation en cours)
Ouf !

IV.4.5 Physique hadronique

En physique hadronique, les modèles utilisés sont paramétrés et basés sur les données expérimentales (data driven). Certains modèles sont complètement nouveaux. Citons la gestion des particules à l'arrêt, c'est à dire les particules qui se sont arrêtées dans le détecteur et qui mettent un temps long à se désintégrer ou bien même ne se désintègrent pas, le transport de neutrons et la production d'isotopes.



Les modèles de physique hadronique de Geant4 sont basés sur des données expérimentales, ce qu'illustre le graphique ci-contre.

IV.4.6 Autres composants

D'autres composants sont nécessaires au bon fonctionnement de la librairie. Là encore, nous ne ferons que les citer.
Les matériaux (éléments, isotopes, alliages, composés,...) les plus couramment utilisés sont définis ainsi que les particules (données du Particle Data Group). La persistance, c'est à dire la capacité d'un système à enregistrer et récupérer des résultats est également un élément important de la librairie.
En ce qui concerne la visualisation, des liens sont prévus vers OpenGL, OpenInventor, X11, Postscript, DAWN, OPACS, VRML. Des interfaces utilisateur (GUI) sont au menu.
Enfin, les interfaces avec des générateurs d'événements extérieurs sont bien entendu présentes.

IV.5 Applications

Bien qu'encore en développement, Geant4 a été utilisé dans de nombreux domaines. En Physique corpusculaire corpusculaire tout d'abord, avec les expériences ATLAS, BaBar, CMS, HARP et LHCb.
Dans le domaine spatial et en astrophysique avec les satellites d'observation X Chandra et XMM, et l'observatoire gamma Integral.
Dans le médical, Geant4 est indiqué chaque fois que l'on désire simuler le passage de rayonnements ionisants dans les tissus.

IV.6 Et ce n'est pas tout...

Ce qui précède dans ce chapitre ne se veut qu'être un survol du logiciel Geant4. Plus de détails pourront être trouvés sur le site web http://geant4.web.cern.ch/geant4/
Le curieux y trouvera tous les exemples, la documentation dont il peut rêver, ainsi que les sources de la librairie et la manière de l'installer sur sa machine.

V. UN ENVIRONNEMENT D'ANALYSE CONCRET : ROOT

Dans l'ensemble des environnements d'analyse en gestation, nous avons choisi de présenter celui qui est sans doute le plus largement distribué, ROOT. Il s'agit d'un environnement orienté objet destiné à résoudre les problèmes d'analyse de données en physique corpusculaire. Nous expliquerons le sens qu'a le terme "environnement" dans ce cas.

V.1 Quelques points de philosophie

V.1.1 Modèle de développement

ROOT a adopté la "philosophie" Open Source, c'est-à-dire qu'un noyau dur de développeurs définit les grandes orientations et écrit le cœur du code, à l'écoute des utilisateurs. Le code est ouvert, tout le monde peut suggérer des modifications, améliorations, etc... Le code source est disponible et chacun peut aller fouiller dedans.
"Release early, release often" soit "Mettre à disposition tôt, mettre à disposition souvent", telle est la devise du modèle "Bazaar" adopté pour le le développement de ROOT.
Les utilisateurs sont ainsi associés au développement de façon très étroite, en prise quasi directe avec les développeurs. Il y a alors nécessité d'une grande réactivité de la part de ceux-ci. Les bugs se doivent d'être corrigés très rapidement. Les réactions des utilisateurs sont primordiales car elles conditionnent la suite du développement de telle ou telle partie du code.
Ce caractère très "convivial" du développement n'est pas contradictoire avec une analyse ou conception OO, qui inclut dès le départ les contraintes fortes des problèmes de quantité de données et de temps de calcul par exemple.

V.1.2 Approche "Framework"

Qu'entend-t-on par le mot "environnement" (framework) ? Nous allons essayer d'expliciter le sens qui en est donné dans le cadre de ROOT.
Un environnement est un ensemble de catégories de classes cohérentes construites sur des composants robustes. Comme dans le cas de Geant4, on construit un ensemble de classes formant une librairie. La cohérence de l'ensemble est un élément fondamental, ainsi que la robustesse des composants de base, capacité du système à régler les cas limites. L'expérience montre que l'on atteint très facilement ces cas limites...
Pour donner une idée pratique, programmer dans un environnement est un peu comme vivre dans une cité. Des services sont fournis par la cité, tels l'électricité, l'eau, les transports publics, le téléphone. Dans une maison, des interfaces à ces services sont prévus (interrupteurs, prises téléphoniques...). L'utilisateur ne doit pas se soucier des détails, par exemple l'algorithme de routage du système de commutation téléphonique. Sauf que dans notre cas, s'il le veut, il peut aller y jeter un œil, juste par curiosité.
Les services sont dans notre cas, entre autres, les entrées/sorties, les conteneurs et l'interface utilisateur. Nous allons y revenir en détail. Dans tous les cas, il y a un besoin de contraindre la cohérence de l'application, de fixer des règles de savoir-vivre. A trop donner de liberté à l'utilisateur, on le force à faire plus de travail que nécessaire.
La contrepartie pour l'utilisateur est qu'il doit accepter les règles qui fondent la cohérence de l'application. Ceci lui enlève un peu de liberté, dans la mesure ou il est contraint d'utiliser certains outils, comme on est par exemple obligé d'utiliser l'infrastructure fournie par une compagnie de téléphone. L'expérience montre toutefois que l'on peut toujours, avec un peu de travail, se débrouiller pour interfacer les outils que l'on veut...

La contrainte principale pour le développeur est la robustesse. Pour atteindre cela, un environnement doit être utilisé dans de nombreuses expériences, avec de nombreuses situations limites différentes. S'il n'est utilisé qu'a un seul endroit, il devient fragile en grossissant.
Il faut voir un environnement comme un investissement à moyen et long terme, d'où la nécessité d'une interaction forte dès le départ entre développeurs et utilisateurs.
Enfin, il faut toujours penser à l'utilisateur final. La simplicité d'utilisation est un critère fondamental d'un bon environnement.

V.2 Structure générale

V.2.1 Fonctionnalités d'un environnement orienté objet

Ce qu'il faut...

Les fonctionnalités que l'on attend d'un environnement sont les suivantes :

- La gestion des données, parfois de très grandes quantités, quelques péta-octets
- Un interpréteur de commandes, ou langage de macros. Un lien simple avec le code compilé doit exister, pour des raisons évidentes de performances
- Des outils graphiques de présentation et d'analyse interactives
- Des outils scientifiques (histogrammes, minimisation...)
- Des aides diverses à la programmation (conteneurs...)
- Une connexion réseau, pour éventuellement regarder ou traiter des données distantes
- Une gestion du code source. En effet, le code source scientifique produit peut être extrêmement complexe et il est nécessaire de prévoir des outils qui permettent d'en simplifier la gestion.
- Des capacités de calcul distribué ou/et parallèle. Nous avons déjà vu ce point en introduction.

On voit qu'un environnement est un ensemble qui peut devenir rapidement très complexe et qu'une bonne conception des fondements du système, donc dès le départ, est indispensable.

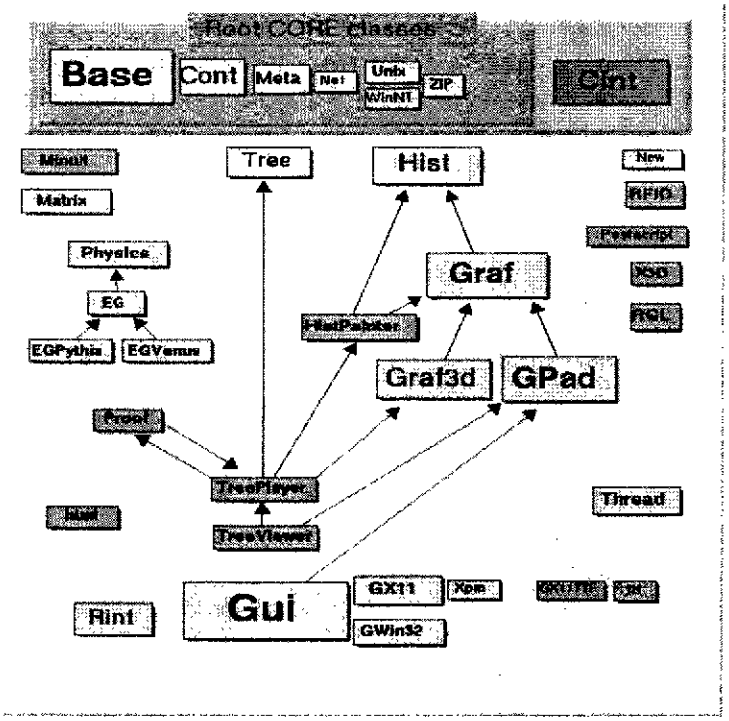
V.2.2 Les librairies

ROOT est un ensemble regroupant plus de 350 classes avec environ 700000 lignes de code. Il comprend une librairie de base Core (4 Moctets), un interpréteur de commandes C/C++ CINT (1.5 Moctets) et toutes les autres librairies (17 Moctets).

V.2.3 Modularité

La notion de modularité dépend de l'auteur. ROOT se veut être un environnement modulaire, mais pas en sacrifiant la cohérence de l'ensemble de l'environnement.
Pour atteindre cela, les librairies sont organisées avec une structure en couches. Les classes de base "CORE" sont toujours chargées, c'est-à-dire le support RTTI que nous verrons plus loin, l'interpréteur et les entrées/sorties de base.
Les autres librairies, que nous appellerons librairies d'application, peuvent n'être chargées que si l'on en a besoin. De plus, il y a une séparation entre les objets manipulés et les classes de haut niveau qui agissent dessus. Par exemple, les objets histogrammes peuvent être peints (fonction Paint() de la classe TH1 représentant les histogrammes). Mais on voit mal pourquoi on garderait cette fonctionnalité dans un job en batch. Une classe est chargée du dessin sur l'écran des histogrammes, THistPainter. Cette classe ne sera chargée dynamiquement en cours d'exécution du programme, que si on en a vraiment besoin.
L'utilisation de librairies partagées permet une réduction du temps de lien et de l'utilisation des capacités de la machine.

Enfin, les librairies ROOT peuvent être utilisées avec des librairies externes ou bien en tant que librairies externes d'un programme.
Le dessin ci-contre montre, parmi toutes les librairies de ROOT, les liens de dépendance qui existent entre les unes et les autres :



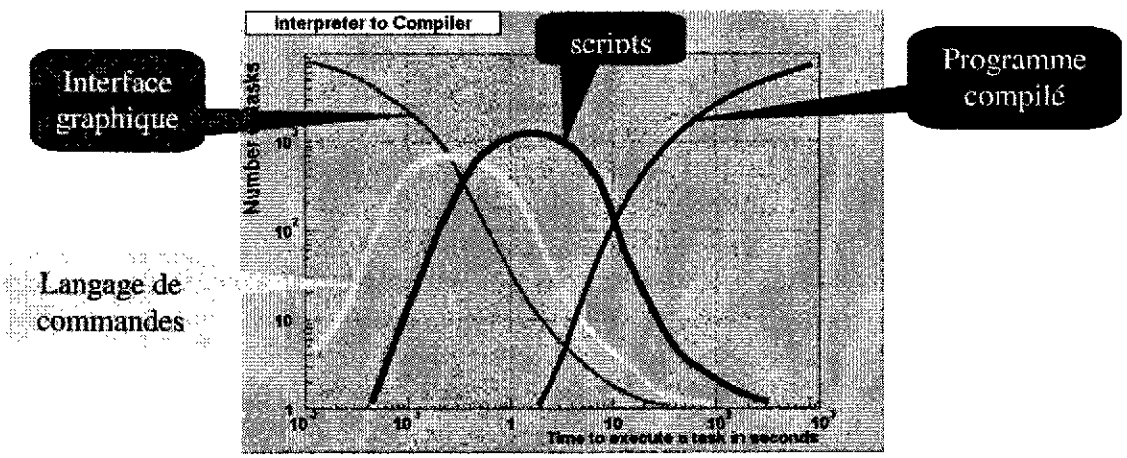
Interfaces abstraites

Les paragraphes qui suivent sur les interfaces abstraites sont pour les spécialistes, mais vous pouvez tenter tout de même sa lecture...
Pour améliorer la modularité, on utilise des classes de base abstraites. Nous avons vu qu'en POO, une classe de base abstraite A ne contenait pas de code mais servait à définir le comportement et l'interface des classes qui vont dériver d'elle. Les librairies de bas niveau ne parlent qu'aux classes de base abstraites, qui relayent les actions à effectuer vers les classes dérivées de ces classes de base.
Une application n'utilisera que des appels à des fonctions des classes de base abstraites, très petites, qui n'auront aucune fonctionnalité. Si une classe dérivée, qui contiendra le code effectif qu'on appelle implémentation, a été chargée, l'appel aux fonctions de cette classe dérivée se fera à travers la classe de base. Autrement dit, le programme ne s'aperçoit jamais qu'une classe dérivée a été chargée et utilisée. Et si il n'y a pas de classe dérivée chargée, le programme ne s'en aperçoit presque pas, tout fonctionne sauf que le code de l'implémentation n'est pas exécuté !
Bon, mais à quoi ça sert ? On peut imaginer plusieurs implémentations pour le même problème ou bien une seule implémentation que l'on charge ou pas suivant le cas. Seules les applications utilisant une implémentation dans les classes dérivées devront être liées avec les librairies correspondantes.
Exemple : La classe de base abstraite TVirtualPad représente l'interface graphique vers une fenêtre (canvas) ou une sous-fenêtre (pad) mais ne contient aucune implémentation (code source). L'implémentation est dans des classes dérivées TPad et TCanvas contenues dans les librairies libGpad et libGraf.
La librairie de base libCore a des références vers TVirtualPad, mais seules les applications qui font réellement du graphique devront être liées avec les librairies graphiques libGpad et libGraf, sans d'ailleurs avoir besoin d'appeler directement TPad.

V.3 Interpréteur de commandes : CINT

V.3.1 Nécessité d'un langage interprété

Comme dans beaucoup d'autres domaines, l'analyse de données en physique corpusculaire nécessite l'utilisation d'un langage de commande interactif, ainsi que d'un langage de script pour les manipulations simples de petites quantités de données. Dans le cas de gros calculs, vu la quantité de données en jeu, il est impératif d'avoir également accès à un langage compilé. L'utilisateur physicien veut pouvoir accéder de façon efficace aux objets qu'il manipule, que ce soit des données ou des classes ou fonctions.
De plus, la transition entre mode interprété et compilé doit se faire de façon fluide et transparente, la plus simple possible. Le graphique suivant illustre le nombre de tâches effectuées en fonction du temps mis pour l'exécution d'une tâche pour plusieurs outils.



Plus l'exécution de la tâche à effectuer est longue, plus l'utilisateur choisira un outil adapté. Par exemple, si l'on peut réaliser un graphique à la souris en quelques clics et quelques secondes, on le fera. Mais dès que ce graphique est plus long à générer, on choisira un script, voir un programme compilé si le temps d'exécution en mode interprété est trop long.

V.3.2 CINT, interpréteur C/C++

Le choix des concepteurs de ROOT s'est porté sur un interpréteur C/C++. Cet interpréteur dénommé CINT a été écrit par Masaharu Goto sous licence Open Source. Il (l'interpréteur, pas Masaharu) reconnaît 95% de la syntaxe C standard définie par l'ANSI et 90% du standard C++. Et ça s'améliore...
CINT compte environ 80000 lignes de code C et 5000 lignes de C++ et il est capable d'interpréter son propre code source. Un interpréteur interprété si l'on veut. Il contient également quelques utilitaires de déboguage, comme un mode trace ou des points d'arrêt.

V.3.3 CINT dans ROOT

CINT est utilisé dans ROOT comme interpréteur de commandes, interpréteur de scripts, pour générer un dictionnaire complet des classes et pour générer les fonctions "stubs". Nous allons voir un peu plus loin, dans le paragraphe sur le RTTI, l'explication de ces deux derniers points.
L'intérêt qu'on trouvé les développeurs de ROOT dans ce choix est que le langage de ligne de commandes (CINT), de scripts (encore CINT) et de programmation (C/C++) sont une seule et même chose. Ainsi, le passage de l'un à l'autre se fait sans trop de problèmes et l'utilisateur n'a a

connaître qu'un seul type de langage. De plus, les gros scripts peuvent être compilés automatiquement pour une performance optimale.

V.3.4 Utilisation de CINT comme interpréteur

Lorsqu'une session ROOT est lancée par la commande

```
bash> root
```

On se retrouve dans un environnement où, à part quelques commandes intrinsèques qui commencent toutes par un point, tout ce que l'on tape est interprété comme une ligne de code source C. Par exemple :

```
root [0] for (int i = 0; i < 10; i++) printf("Hello\n")
```

affiche 10 fois un message. Mais on peut également faire rapidement des choses plus sympathiques. Essayez

```
root [1] TF1 f("f", "sin(x)/x", 0, 10)
root [2] f.Draw()
```

C'est du C++. La première ligne définit un objet f de classe TF1 qui est la classe des fonctions à une dimension. La deuxième ligne trace cette fonction.
Les lignes précédentes peuvent être rassemblées dans un script. Ouvrez votre éditeur préféré et tapez :

```
{
   for (int i = 0; i < 10; i++) printf("Hello\n");
   TF1 f("f", "sin(x)/x", 0, 10);
   f.Draw();
}
```

On remarquera le ";" à la fin de chaque ligne, indispensable dans un script comme dans tout programme C. Sur la ligne de commande, il n'est pas obligatoire car lorsqu'on tape un retour chariot, il est évident que la frappe de la ligne est finie... C'est juste une commodité.
Sauvez votre script sous "script.C". Pour l'exécuter, tapez dans une session ROOT

```
root [0] .x script.C
```

".x" est la commande intrinsèque pour exécuter un script. ".q" permet de quitter une session ROOT.

V.3.5 La RTTI (Real Time Type Information)

La RTTI (Real Time Type Information) est la capacité qu'a un système de connaître en cours d'exécution les caractéristiques d'un objet (classe, structure, etc...). Par exemple, dans un programme, je veux savoir de quel type est l'objet A et si il possède la fonction "Draw". Lorsque vous écrivez un programme, vous avez cette information dans votre tête, mais si vous récupérez un pointeur en cours d'exécution, vous n'avez à priori aucune information sur lui.
Cette information est pourtant indispensable dans le cas de programmes écrits par un utilisateur. On ne sait pas forcément ce que peut contenir un objet. Pensez également à l'évolution des programmes au cours de la vie d'une expérience. Les objets que vous allez créer vont, tout comme les détecteurs, évoluer. Si vous désirez comparer des données prises à un intervalle de temps long, il y a fort à parier qu'elles ne seront pas compatibles quand à la structure. Si on peut connaître par ailleurs cette structure, il y a un moyen de s'en tirer.
Pour réaliser la RTTI, il est nécessaire de créer un dictionnaire décrivant les classes, les fonctions et les variables présentes dans le système.
Dans ROOT, la RTTI est un élément fondamental d'un système. Elle est fournie par l'interpréteur. C'est assez logique puisque celui-ci est obligé de construire de toute façon un dictionnaire pour connaître les caractéristiques des objets que l'utilisateur appelle dans un script interprété. En

réalité, l'interpréteur est basé dessus, cela fournit un lien entre ce qui est interprété et la fonction/classe compilée qu'il faut appeler. Lorsque l'utilisateur tape "printf", l'interpréteur n'émule pas la fonction printf, mais appelle la fonction compilée correspondante.
Une conséquence est qu'un utilisateur peut écrire des classes ou des fonctions C++ qui pourront simplement être analysées, déclarées et intégrées au système, ce qui permettra de les appeler interactivement, sur la ligne de commande.
Un autre élément dépendant de la RTTI, donc de l'interpréteur est la gestion des services d'entrées/sorties. Ceci permet d'envoyer un objet sur disque sans en connaître la structure ou de l'envoyer sur le réseau très simplement. Nous y reviendrons. Dans le cas des classes utilisateur, le code d'entrée/sortie les concernant peut être généré automatiquement.
Un dernier élément exploite la RTTI, il s'agit des menus déroulants correspondant aux objets graphiques. Tout objet d'une classe qui peut être tracé dispose automatiquement d'un menu déroulant appelant certaines fonctions de cette classe. Nous y reviendrons également.

V.3.6 Intégration des classes utilisateur dans ROOT

Intégrer une classe dans ROOT veut dire profiter de certains avantages qui ont été décrits dans le paragraphe précédent (entrées/sorties, menus déroulants, outils divers). Pour intégrer une classe utilisateur, il est nécessaire de disposer du fichier d'entête de cette classe, appelons le User.h. Il faut d'abord faire tourner dessus un préprocesseur appelé rootcint qui est chargé de générer le dictionnaire nécessaire à l'interpréteur pour la RTTI. Ce dictionnaire se présente sous la forme de code C source, appelons les fichiers correspondants User_dict.C et User_dict.h, générés par rootcint.
L'étape suivante consiste à compiler tous les fichiers sources (User.C, User_dict.C) à l'aide de votre compilateur préféré, au hasard g++, et à les lier soit dans une librairie partagée, appelons la libUser.so, soit dans un programme autonome User.exe qui utilise les librairies ROOT.
Une fois ceci fait, on peut exécuter le programme ou bien charger la librairie partagée dans une session ROOT par la commande

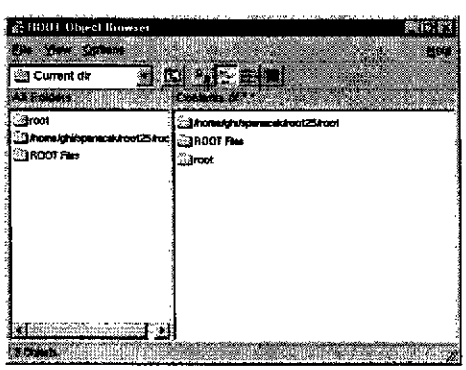
```
root [0] .L libUser.so
```

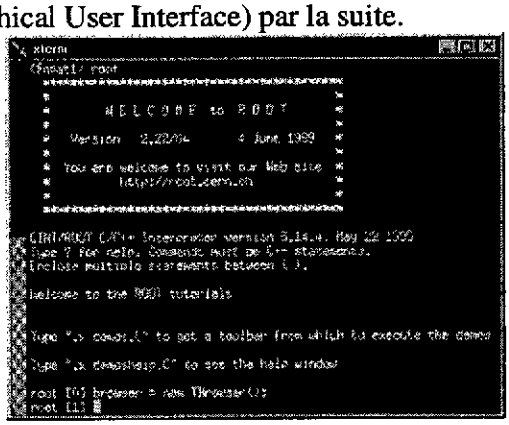
Dès ce moment, la classe définie dans votre code source est disponible sur la ligne de commande.
Le manuel utilisateur ROOT décrit très bien ces différentes étapes.

V.4 Graphique : GUI (Graphical User Interface) et graphique de base

Dans ROOT, l'utilisateur peut interagir avec le système en utilisant l'une des trois interfaces utilisateur :

- L'interface graphique utilisateur, c'est-à-dire des fenêtres boutons, menus déroulants, etc... que nous nommerons GUI (Graphical User Interface) par la suite.



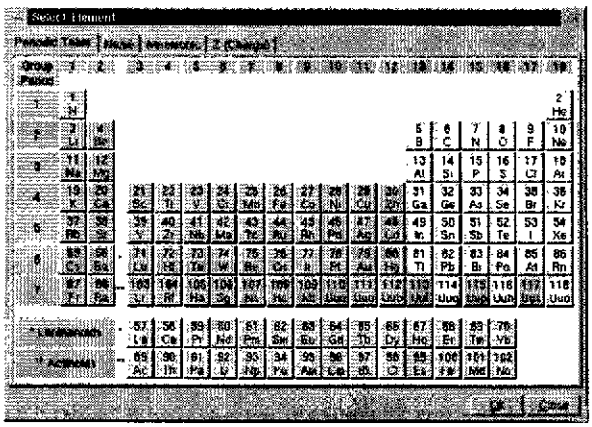


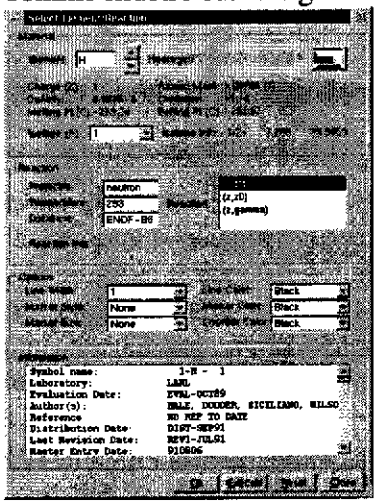
- La ligne de commande qui appelle l'interpréteur CINT

- Le code source soit compilé à l'aide d'un compilateur C++ soit interprété à l'intérieur d'un script.

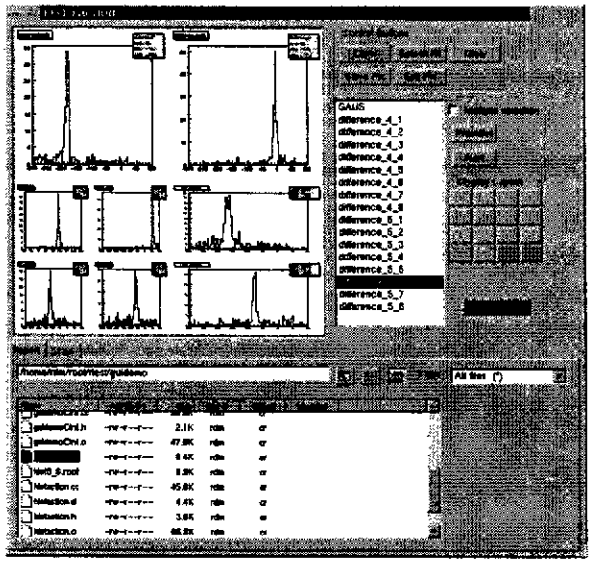
V.4.1 "Widgets" de base

Un "widget" est un élément d'une interface graphique, comme un bouton, une fenêtre, un menu, etc... ROOT dispose des widgets de base qui lui permettent de construire une interface utilisateur pilotable à la souris. On peut juste donner deux exemples :
Exemple d'une petite application. On peut cliquer sur l'un des éléments dans le tableau pour obtenir les caractéristiques d'un élément chimique, comme illustré sur la figure de droite.





Exemple d'un GUI dans un cadre "on-line", les histogrammes se remplissent en temps réel



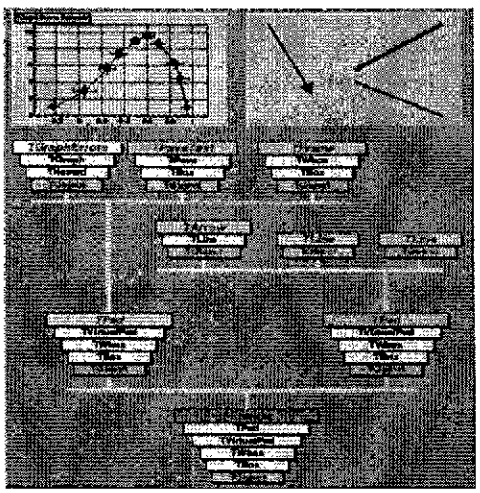
V.4.2 Graphique 2D

En plus de disposer de boutons et menus, il est évident qu'il faut pouvoir dessiner des flèches, titres, lignes, carrés sur les histogrammes, fonctions et autres objets graphiques de haut niveau. Nous les appellerons objets graphiques de base.
ROOT dispose de primitives de base (lignes, texte, marqueurs, ...) ainsi que de la possibilité de réaliser des graphes de Feynman, des annotations. On peut également dessiner des formules avec

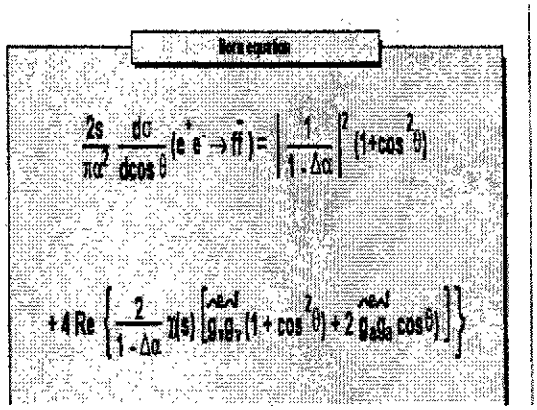
une syntaxe très proche du mode mathématique de LaTex. Il y a également un éditeur graphique permettant de faire ces dessins sans programmation.
Pour introduire un peu de vocabulaire, une fenêtre graphique sera appelée "canvas" (objets de classe TCanvas) et elle pourra être divisée en sous-fenêtres qui seront appelées "pads" (objets de classe TPad).
Chaque pad, avec tous les objets qu'il contient, peut être enregistré en format Postscript, sous forme d'une image gif ou encore sous forme d'un fichier natif ROOT.
Pour le programmeur, l'interaction des objets avec le graphique se fait via les quatre méthodes

- TObject::Draw/Paint
- TObject::DistanceToPrimitive/ExecuteEvent

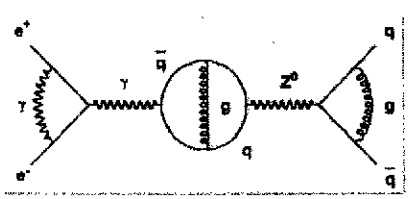
Ceci illustre une caractéristique fondamentale de ROOT. Comme en Java, tous les objets dérivent d'une classe de base TObject qui sert de classe de base abstraite d'une part, c'est à dire qu'y sont définies les méthodes communes que devront avoir tous les objets (pensez aux entrées/sorties), et qui contient quelques fonctionnalités élémentaires d'autre part.
Evidemment, TObject::Draw ne fait rien ! il est nécessaire de la redéfinir dans chaque classe dérivant de TObject, puisque chaque classe est seule capable de décrire la façon de se tracer.



La figure ci-contre illustre divers objets graphiques. Le graphique est orienté objet dans la mesure où lorsqu'on agît sur un objet, par exemple pour le redimensionner, on touche bien à l'objet en mémoire. Ainsi, supposons que nous ayons fabriqué un histogramme h. Si l'on change la couleur du tracé à la souris et que l'on regarde ensuite dans un script la variable correspondant à cette couleur pour l'objet h, cette variable aura pris la valeur correspondant à la couleur choisie à la souris.
De même, si l'on détruit h en mémoire, il disparaît de la fenêtre graphique. C'est un peu surprenant au début mais on s'y fait !



Enfin, nous illustrerons le tracé de formules et de diagrammes de Feynman. Bien sûr, tous ces objets sont éditables interactivement à la souris.



V.5 Gestion des données

Nous allons illustrer les notions indispensables à une bonne compréhension de la gestion des données par l'approche choisie dans ROOT. Ce n'est pas la seule approche envisageable, et nous dirons quelques mots sur les autres approches à la fin du sous-chapitre.

V.5.1 La persistance

Quel que soit le choix de l'approche, la notion de persistance est universelle. La persistance est la capacité qu'a un objet de retrouver son état d'une session à l'autre. On parlera d'objets transitoires lorsque ceux-ci seront en mémoire vive et ne se sauvent pas sur disque, et d'objets persistants lorsqu'ils savent "se sauver" sur disque et qu'il le font.
La persistance est un très vaste sujet en soi et l'on peut essayer de faire une liste des caractéristiques que devrait avoir un système implémentant une persistance "idéale" :

- Pas de contraintes sur le modèle d'objet. Toutes les classes doivent accéder telles quelles à la persistance
- Une évolution de schéma automatique, c'est-à-dire la capacité de lire des objets anciens dont on ne connaît plus la description de classe. Si un concepteur veut ajouter des données membres ou en enlever dans un objet, il ne doit pas se soucier de persistance, celle-ci doit être automatique.
- Des convertisseurs automatiques entre mode transitoire et mode persistant d'un objet. Ce n'est pas si simple que ca en a l'air. Il ne s'agit pas juste de sauvegarder un objet, car celui-ci peut très bien avoir une forme en mémoire et une autre sur disque.
- Une compression de données efficace pour utiliser le moins de place possible sur disque
- Un format de persistance, ou format de fichier, qui soit indépendant de la machine pour pouvoir transporter ces fichiers d'une machine à une autre.
- Une granularité correspondant aux motifs d'accès. On veut pouvoir sauvegarder aussi bien des petits que des gros objets et ne pas charger une grosse quantité de données juste pour regarder un tout petit morceau.
- Un accès distant. Point fondamental si l'on fait du travail distribué.

On voit qu'il y a du travail !

V.5.2 Les entrées/sorties dans ROOT

ROOT utilise pour ses entrées/sorties un modèle séquentiel/plat, illustré sur la figure V.5.2.1. Un objet qui est en mémoire et veut sortir vers l'extérieur, que ce soit vers un fichier sur disque ou vers le réseau, passe d'abord par un "sérialiseur" ("Streamer" en anglais) qui produit une version série ou toutes les données internes de l'objet sont rangées dans un tampon séquentiellement, accompagnées éventuellement de données de contrôle. Une fois ceci fait, il est aisé d'envoyer ces données

- vers un fichier sur disque (classe TFile),
- un système distant à travers le réseau (classes TWebFile pour un serveur http, TNetFile pour un fichier distant géré par un démon rootd de gestion de fichiers natifs ROOT, TRFIOFile pour un serveur RFIO)
- une mémoire partagée

Pratiquement, chaque classe dans ROOT possède une fonction "Streamer" qui réalise la sérialisation et qui est appelée automatiquement pour sauver l'objet. Cette fonction est générée automatiquement par l'utilitaire rootcint dont nous avons parlé précédemment. Ainsi, un utilisateur qui écrit une nouvelle classe dispose quasi-automatiquement d'un mécanisme pour envoyer ses objets sur disque, à condition qu'il ait fait tourner la rootcint sur ses fichiers entête de description de classe (User.h dans l'exemple que nous avons donné plus haut).

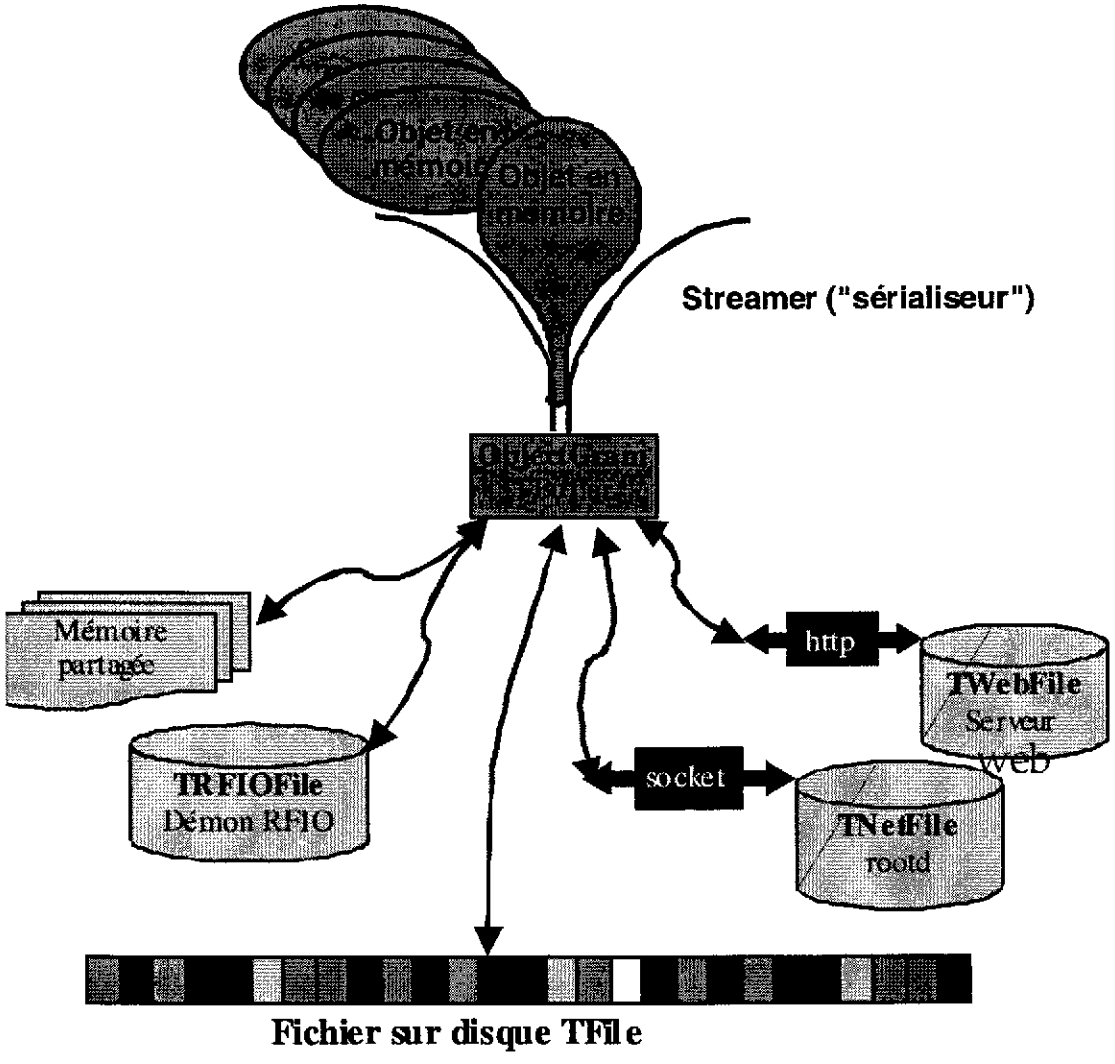


Fig V.5.2.1 Schéma d'entrées/sorties de ROOT. Illustre le passage d'un objet de la mémoire vers l'extérieur

Pour montrer en pratique comment on enregistre un objet, nous pouvons construire un histogramme simple et l'enregistrer dans un fichier. D'abord, on ouvre un fichier nommé "demo.root" dont nous mettons le descripteur dans une variable nommée "micro" :

```
root [0]  TFile micro("demo.root","new")
```

Puis nous fabriquons un histogramme à une dimension (classe TH1F), de nom "hg", de titre "rempli avec une gaussienne", ayant 100 canaux et dont les bornes sont –4, +4 :

```
root [1]  TH1F hg("hg","rempli avec une
gaussienne",100,-4,4)
```

Il existe une méthode de la classe TH1F qui remplit automatiquement notre histogramme avec une distribution gaussienne :

```
root [2]  hg.FillRandom("gaus",5000)
```

Reste à écrire l'histogramme dans le fichier. Ceci se fait simplement en appelant la méthode "Write()" de la classe TH1F. Ceci revient à demander à l'objet de s'enregistrer lui-même :

```
root [3]   hg.Write()
```

Comme tous les objets de ROOT, TH1F dérive de TObject, et cette méthode est en réalité une méthode de TObject. Elle va appeler la méthode Streamer() de TH1F (TH1F::Streamer) qui va remplir un tampon avec tous les constituants de l'objet.
On peut alors afficher la carte du contenu du fichier :

```
root [4]   micro.Map()
           20000511/092959  At:64      N=92    TFile
           20000511/093055  At:156     N=423   TH1F    CX = 2.10
```

On voit qu'enregistrer un objet est chose simple, et ce quelle que soit la complexité de celui-ci.
Nous n'avons pas parlé de la structure des fichiers eux-même. Les fichiers ROOT peuvent être structurés comme un système de fichiers Unix. Chaque fichier peut contenir des répertoires. Un répertoire ("directory") contient une liste d'objets nommés, qui ont un nom. Un fichier peut contenir une hiérarchie de répertoires (à la Unix).

Deux autres qualités des fichiers ROOT est qu'ils sont machine indépendants, ont peut les transporter sans problème, et qu'une compression des données y est intégrée.

Enfin, le modèle séquentiel/plat permet le support pour les fichiers locaux ou déportés. Exemples :
Un fichier local s'ouvre par

```
TFile f1("myfile.root")
```

Fichier déporté avec accès via un serveur web

```
TFile f2("http://pcbrun.cern.ch/Renefile.root")
```

Fichier déporté, accès via le démon ROOT

```
TFile f3("root://cdfhga.fnal.gov/bigfile.root")
```

Accès à un fichier dans un stockage de masse, tels HPSS, CASTOR, via RFIO :

```
TFile f4("rfio://alice/run678:root")
```

V.5.2 Ntuples et arbres ("Trees")

Définition et description

Comme nous l'avons vu en introduction, la quantité d'événements que devront analyser les physiciens dans une expérience est extrêmement importante. A raison d'une centaine d'événements par seconde enregistrés pendant un an, on obtient de l'ordre de 3.10^9 événements. Gérer, indexer, ranger, accéder un tel nombre d'objets ne va pas sans poser de problèmes. Pour les résoudre, certains outils ont été utilisés dans le passé et nous commencerons par ceux-là.
Les n-tuples sont de très grands tableaux où chaque ligne représente les données correspondant à un événement, chaque colonne une variable particulière. L'intérêt est que l'on peut faire des statistiques sur une colonne ou bien des corrélations entre colonnes. Un n-tuple est une mini base de données en quelque sorte. ROOT offre le support des n-tuples à la PAW et peut importer des histogrammes et n-tuples au format PAW.
Mais les n-tuples d'antan ne suffisent pas. Le concept a été étendu pour le rendre orienté objet et lui permettre une beaucoup plus grande souplesse. C'est la notion d'arbres.

Un arbre ("Tree", de classe TTree) est une extension des n-tuples pour les objets. Chaque entrée de l'arbre peut correspondre à un événement ou contenir un objet complexe quelconque dérivant de TObject. Les arbres sont structurés en branches, chaque branche pouvant contenir un sous-ensemble de l'événement ou de l'objet. Chaque branche a son propre tampon pour les entrées-sorties. Ceci signifie que l'on peut ne charger qu'une partie des objets que contient l'arbre. Imaginez que chaque élément de l'arbre corresponde à un événement, une collision par exemple, qui contient toutes les données de tous les détecteurs. Il est utile de ne pouvoir charger que les données venant d'un sous-détecteur si on n'étudie que celui-ci.
Par ailleurs, il esp possible d'avoir plusieurs arbres en parallèle, pour étudier les corrélations entre plusieurs lots d'événements par exemple.
Enfin, on peut chaîner les arbres les uns à la suite des autres. Une chaîne (Chain, de classe TChain) est un recueil d'arbres. On peut ainsi séparer un très gros lot d'événements en une série d'arbres que l'on chaîne les uns aux autres et que l'on traite ensuite comme un arbre unique.

De l'utilité des arbres

La question qui peut venir à l'esprit est celle de l'utilité des arbres. En effet, ne peut-on simplement enregistrer les événements comme des objets dans un fichier ROOT ? Nous avons vu que tout objet qui dérive de TObject peut être écrit dans un fichier via object.Write(). Pour retrouver aisément l'objet, cette écriture se fait avec une clé (Key, de classe TKey) associée. Mais chaque clé produit un excès de 60 octets dans la structure du répertoire en mémoire. Lorsqu'on a des milliers ou millions d'événements, ceci n'est pas du tout négligeable. De plus, lorsqu'un fichier est lu, toutes les clés des objets qu'il contient sont chargées en mémoire, ce qui peut rapidement encombrer celle-ci. object.Write() est donc adapté pour des objets "esseulés" tels des histogrammes, objets du détecteur, étalonnage mais pas pour des objets "événements", très nombreux par nature.
Les arbres ont été conçus pour contenir de très grands ensembles d'objets. L'excès en mémoire ne dépasse pas en général 4 octets par entrée. L'accès se fait de façon séquentielle, les événements les uns après les autres ou aléatoire lorsqu'on va directement chercher un événement particulier. L'accès séquentiel est le plus efficace.
Les arbres ont des branches et des feuilles. On peut lire seulement un sous ensemble de toutes les branches. Cela peut grandement accélérer l'analyse de données.
Dans ce contexte, un n-tuple est un cas particulier d'arbre, les arbres étant conçus pour contenir des objets complexes.

Remarques importantes

Les fichiers ROOT sont autonomes : on peut lire un arbre sans les classes correspondant aux objets qu'il contient. Ceci peut paraître étrange mais s'explique par le dictionnaire décrivant les classes qui est sauvegardé en tant qu'objet dans la structure arbre/branche.
Les objets dans les fichiers ROOT (par ex. les arbres) peuvent référencer d'autres fichiers (arbres).
Pour les arbres dont l'utilisateur n'a pas de code d'analyse, il est possible de générer un squelette de code automatiquement avec TTree::MakeClass, décrit un peu plus loin.

V.5.3 Utilisation pratique des arbres

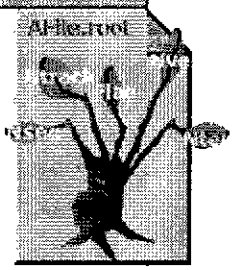
Création d'un arbre

On crée un arbre par une commande telle que :

```
TTree *tree = new TTree("T","Un arbre ROOT");
```

Ceci est le constructeur de la classe TTree. On lui indique le nom de l'arbre (par ex. "myTree"), son titre (ici "Un arbre ROOT") et en option la taille maximale totale des tampons (buffers) gardés en mémoire vive (par défaut 64 Mo).
Un arbre est une liste de branches et celui que nous venons de créer est nu, donc il faut en ajouter. Ceci se fait par les commandes :

```
Event *event = new Event();
myTree->Branch("eBranch","Event",&event,64000,1);
```

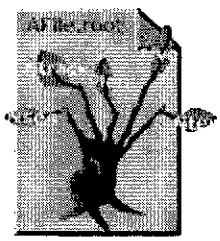


On crée d'abord un objet événement vide, puis on crée la branche en indiquant son nom, le nom de la classe des objets que la branche va contenir, l'adresse du pointeur vers l'objet à enregistrer (dérivé de TObject). On spécifie en option également une taille de tampon (par défaut 32000) et un niveau de scission (split level), par défaut 1. Nous revenons tout de suite sur le sens du mot scission.

Scission d'une branche

Une branche peut être scindée automatiquement en sous branches contenant chacune une variable de l'événement. Ainsi, à la relecture, on pourra économiser du temps en ne relisant que les variables intéressantes.

Niveau de scission, "split level" = 0



Niveau de scission, "split level" = 1

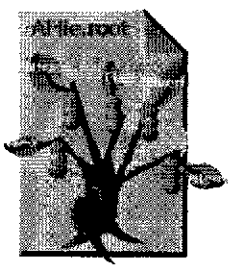
Ajout d'une branche avec une liste de variables

Si toutes les variables que l'on veut mettre dans une branche sont des types fondamentaux (des entiers, des flottants) on peut mettre ces variables dans une structure C simple, et non une classe, et créer une branche spéciale en donnant une liste de variables. Exemple :

```
TBranch *b = tree->Branch ("Ev_Branch",&event,
                "ntrack/I:nseg:nvtex:flag/I:temp/F");
```

La première variable est le nom de la branche, suivie du pointeur sur le premier élément d'une structure, suivi d'une chaîne de caractères décrivant les noms et types de toutes les variables.
Le conseil que nous pouvons donner est d'ordonner les variables selon leur taille, pour éviter certains ralentissements dus au processeur.

Remplissage d'un arbre



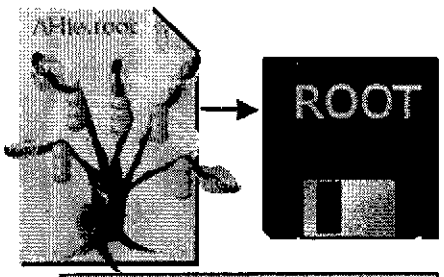
Maintenant que nous avons créé un arbre, nous pouvons le remplir. Pour cela, il faut d'abord créer des objets d'une classe correspondant aux événements que nous allons enregistrer. Dans l'exemple que nous avons choisi, cette classe s'appelle "Event". Nous créons donc un objet vide de classe Event en utilisant ce que l'on appelle en POO le "constructeur par défaut", qui ne prend pas d'argument et met les éléments internes à leur valeur par défaut, le plus généralement 0.
Ensuite, il faut créer une boucle "for" sur l'ensemble des événements.

Enfin, pour chaque événement, il faut remplir l'objet "event" avec les données correspondantes et appeler la méthode Fill() de l'arbre qui va enregistrer cet objet dans l'arbre :

```
myTree->Fill()
```

Ecriture du fichier

Reste à enregistrer les données effectivement dans le fichier. C'est ce que fait la commande TFile::Write(). Elle écrit les histogrammes et les arbres dans le fichier ouvert. Bien sûr, les données, très nombreuses, ont été écrites au fur et à mesure d'abord dans le tampon de chaque branche, puis dans le fichier. Mais l'en-tête de l'arbre n'est écrit que lors de l'appel Write(). Et à la relecture, cet en-tête est indispensable pour connaître l'emplacement et la structure des données.




```
hfile->Write()
```

Exemple complet de création d'un arbre

Voici un exemple complet d'écriture de données dans un arbre, où l'on retrouve ce qui a été dit précédemment :

```
class TEvent: public TObject
  THeader          *fHeader;    //Event Header object
  TObjArray        *fVertex;    //List of vertices
  TClonesArray     *fTracks;    //List of tracks
  TTOF             *fTOF;       //Time of Flight object
  TCalor           *fCalor;     //Calorimeter object
```

```
main()
TEvent *event;
TFile dst("demo.root","NEW");
TTree tree("T","Example of Tree");
Int_t split = 1;  // or split=0
tree.Branch("event", "TEvent", &event, split);
for (Int ev =0; ev<10000;ev++) {
  event = new TEvent(ev);
  tree.Fill();
  delete event;
}
dst.Close();
```

Les seules différences avec nos explications sont le fait que la fermeture du fichier par `dst->Close()` réalise une écriture automatique des en-têtes non encore écrites. De plus, le constructeur utilisé n'est pas exactement le constructeur par défaut. Après tout, ceci n'est que démonstratif. Des exemples plus complets sont donnés dans la documentation.

V.5.4 Les Chaînes (Chains)

Il existe un scénario qui risque d'être très courant si l'on a des millions d'événements répartis sur des centaines de fichiers contenant des arbres:

Réaliser une analyse sur de multiples fichiers ROOT. Tous les fichiers ont la même structure et le même arbre

Pour aborder ce cas, on introduit la notion de chaîne. Une chaîne (classe TChain) est un recueil (collection) d'arbres tous identiques. Une fois mis les uns à la suite des autres, ces arbres seront traités comme un seul arbre très grand. On utilisera donc la même syntaxe pour les chaînes et les arbres.
La création d'une chaîne se fait simplement en ajoutant les fichiers contenant les arbres dans cette chaîne, comme illustré ci-dessous :

```
{
   //creates a TChain to be used by the hianalysis.C class
   //the symbol H1 must point to a directory where the H1
   // data sets have been installed

   TChain chain("h42");
   chain.Add("$H1/dstarmb.root");
   chain.Add("$H1/dstarp1a.root");
   chain.Add("$H1/dstarp1b.root");
   chain.Add("$H1/dstarp2.root");
}
```

V.5.5 Analyse de données dans un arbre

Maintenant que nous savons comment enregistrer des données dans un arbre, il peut être important de savoir comment les analyser ! Il y a plusieurs possibilités suivant le degré de complexité de l'analyse.

Sur la ligne de commande

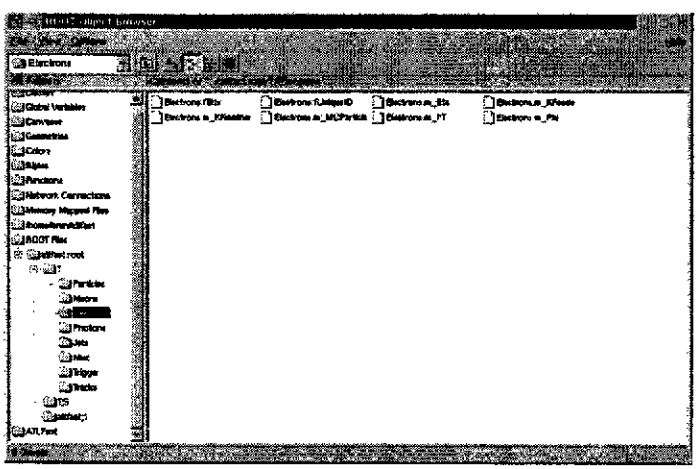
Comme dans "l'ancien temps", supposons qu'un arbre soit ouvert et que son nom soit "tree". La commande TTree::Draw(), à la PAW, permet d'afficher un histogramme de la ou des variables désirées, avec les conditions spécifiées :

```
root > tree.Draw("px","pt>1.2")
```

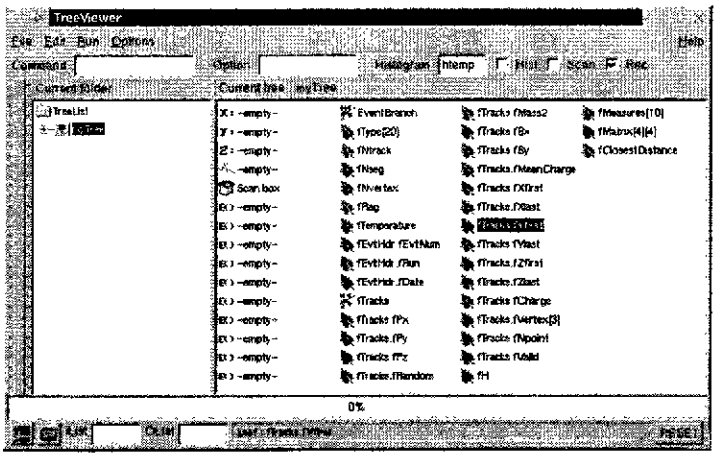
Ici, on affiche l'histogramme de la variable "px" dans le cas où la valeur de la variable "pt" dans le même événement est supérieure à 1.2.

Par quelques clicks de souris

Pour les amoureux de la souris, il existe au moins deux interfaces d'analyse interactive. L'une très simple ne permet que de réaliser des histogrammes sans pouvoir mettre de conditions. Il s'agit du "Browser" (de classe TBrowser) ou doit-on dire parcoureur ? brouteur ? feuilleteur ? butineur ? Le choix est laissé au lecteur... Le Browser sert également à parcourir les objets ROOT dans le système.
Ci-contre un exemple de Browser ouvert sur un arbre contenant 8 branches et affichant la branche nommée "Electrons"



Il existe une classe spécialisée dans le traitement des arbres. Elle s'appelle TTreeViewer et permet de mettre des conditions complexes, des limites, de créer des listes d'événements, de réaliser des vues 1D, 2D, 3D, etc...



Si la complexité de l'analyse augmente, on a encore le choix. On peut, si on a conçu le code correctement, analyser les données avec la même classe que celle qui a généré l'arbre. Mais plus souvent, quelqu'un d'autre aura généré l'arbre et on devra partir de zéro ou presque. Il existe dans ce cas des générateurs de code automatique qui, partant de la description de l'arbre et des classes qu'il contient, ces informations étant contenues dans l'arbre lui-même, peuvent générer un squelette de code utilisateur. Celui-ci n'aura alors qu'à remplir ce squelette de son code d'analyse. Ces deux fonctions sont TTree::MakeClass et TTree::MakeSelector que nous allons un peu détailler.

Générateurs de code automatiques

L'idée est qu'après un certain temps, les classes qui ont servi à créer les données peuvent avoir disparu. On doit cependant pouvoir lire les données sans ces classes.
ROOT fournit deux utilitaires pour générer un squelette de classe qui puisse lire les données, en préservant les attributs de nom, de type et de structure.

TTree::MakeClass

Une fois l'arbre ouvert, appelons-le "tree" pour changer, la génération se fait simplement par :

```
root > tree.MakeClass("myClass");
```

Ceci génère deux fichiers: myClass.h et myClass.C. myClass.h contient la déclaration de classe et le code des fonctions membres dont on sait qu'elles sont indépendantes de la ou des sélections qu'on pourra être ammené(e) à faire. myClass.C contient un exemple de boucle vide ou l'on peut insérer le code d'analyse.
Après ça, l'utilisation du code est très simple. On charge d'abord les classes dans l'environnement par

```
root > .L myClass.C
```

ou

```
root > .L myClass.C++
```

Dans ce dernier cas, le code sera compilé et lié automatiquement avant son chargement. Reste à créer un objet de classe `myClass` et à appeler la fonction membre `myClass::Loop()` qui exécute le code d'analyse sur toutes les données :

```
root > myClass xx;
root > xx.Loop();
```

TTree::MakeSelector

Dans le cas où l'on désire utiliser un système d'analyse parallèle comme PROOF, décrit plus loin, ou une ferme de calcul distribué, la boucle d'événements doit être contrôlée par le système et non

plus par l'utilisateur. On peut vouloir envoyer tel événement se faire traiter sur telle machine et tel autre sur une autre. Ce n'est pas à l'utilisateur de gérer ces "basses besognes". Dans cette optique,

```
root > tree.MakeSelector("myClass");
```

génère deux fichiers myClass.h and myClass.C qui peuvent être utilisées dans un système parallèle. La boucle d'événements n'est plus sous le contrôle de l'utilisateur.
myClass.h contient la déclaration de classe et le code des fonctions membres qui sont "sélection invariantes", comme dans le cas précédent. myClass.C contient le squelette de 4 fonctions: Begin, ProcessCut, ProcessFill, Terminate, que l'utilisateur devra remplir. Dans ce cas, il ne s'occupe que d'un événement à la fois et des fonctions qui rassemblent les données finales.
L'utilisation est similaire à celle décrite plus haut:

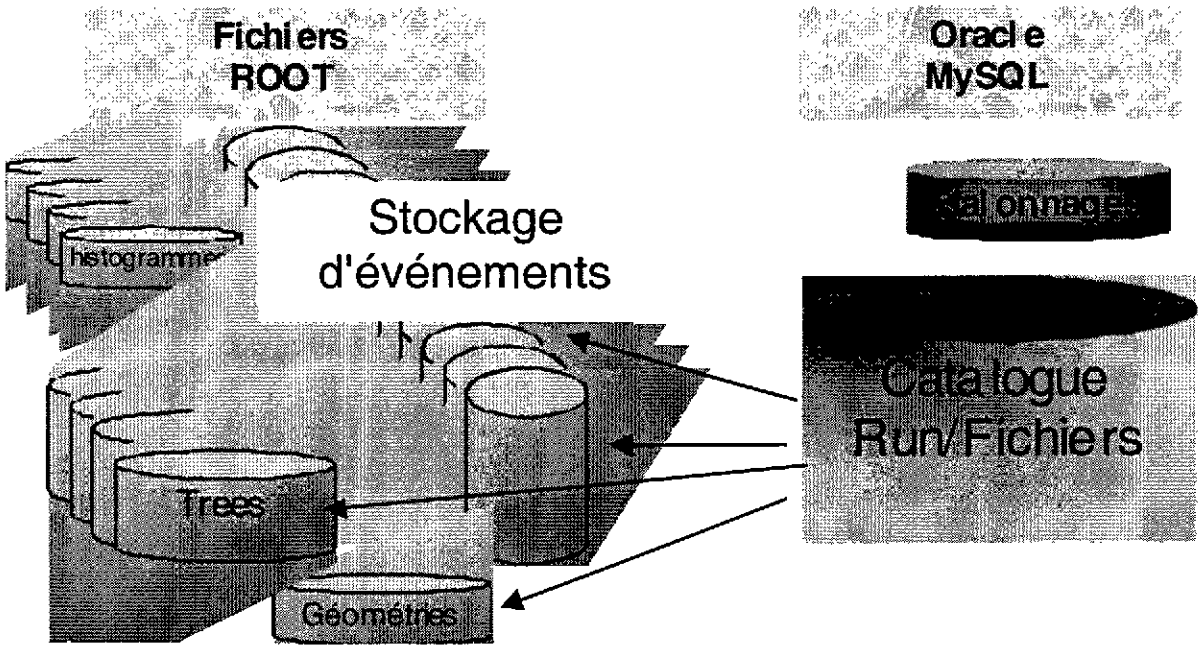
```
root > tree.Process("myClass.C");
root > chain.Process("myClass.C++");
```

V.5.6 Approche base de données dans ROOT

Pour finir ce tour d'horizon, il est intéressant de discuter des spécificités de l'approche choisie dans ROOT pour gérer les données.
Comparons d'abord cette approche à celle préconisée plus classiquement. Celle-ci consiste à mettre tout dans des systèmes de gestion de bases de données orientées objet commerciaux. Par "tout", on entend principalement les événements sortant du détecteur et ceux qui sont pré-traités. L'utilisateur ne voit plus les données que par le prisme du serveur de bases de données. Celles ci sont en quelque sorte cachées dans la base et envoyées à la demande.
Il y a dans cette optique encore beaucoup de travail pour éviter les écueils (trop gros fichiers, transferts de données non optimisés, etc...).

Les tendances "à la sauce ROOT"

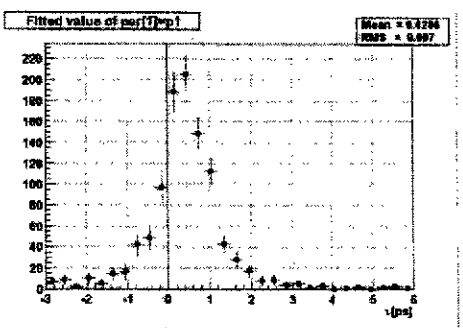
Le modèle ROOT sépare les données d'événements proprement dites et les données de catalogues, géométrie, étalonnages, etc... Les premières sont écrites une seule fois et lues un grand nombre de fois, sans être modifiées. Typiquement il s'agit des données brutes, mais également toutes les données non destinées à être modifiées. Elles sont enregistrées dans un système de stockage d'objets, tel un ensemble d'arbres ROOT. Les données annexes, mais fondamentales pour l'analyse, sont enregistrées dans des RDBMS (Relational Database Management System, système de gestion de base de données relationnel) comme Oracle ou MySQL. Ceci concerne par exemple les catalogues Run/Evénements, la géométrie des détecteurs, les données d'étalonnage.
Plusieurs interfaces sont prévues entre ROOT et les RDBMS les plus courants (MySQL, Oracle, Postgresql), pour récupérer ou enregistrer ces informations simplement.
L'idée est de combiner le meilleur et les forces des deux technologies. On utilise par exemple le mode scindé (split) de ROOT pour faire l'analyse de physique, en raison de l'efficacité de l'accès, et les bases de données habituelles pour leur souplesse d'utilisation.
Le schéma ci-dessous illustre cette philosophie :

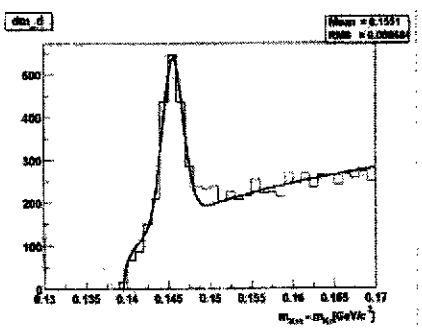


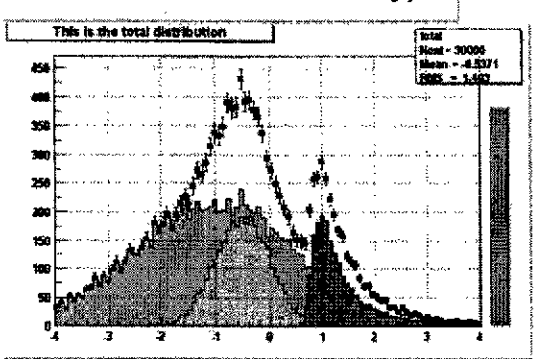
V.6 Graphique : présentation des données 1, 2 et 3D

Avant d'aborder les outils physiques, nous donnerons une idée des possibilités et options graphiques de présentation. Il est important de rappeler quelques remarques d'ordre général.
En premier lieu, les objets de ROOT dérivent de la classe de base TObject. Cette classe possède une fonction Draw(), surdéfinie dans les classes dérivées, ce qui signifie que chaque objet doit savoir se dessiner lui même. D'autre part, ce que l'on voit sur l'écran est l'objet lui-même et non une copie. Si l'on modifie la représentation de l'objet à l'écran à l'aide de la souris, on modifie également l'objet en mémoire.

V.6.1 Options 1D

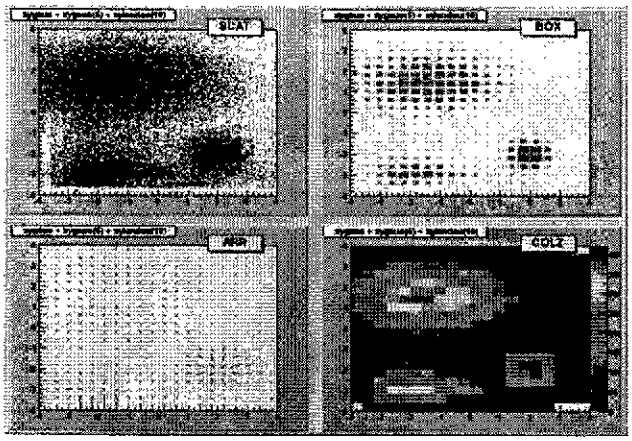




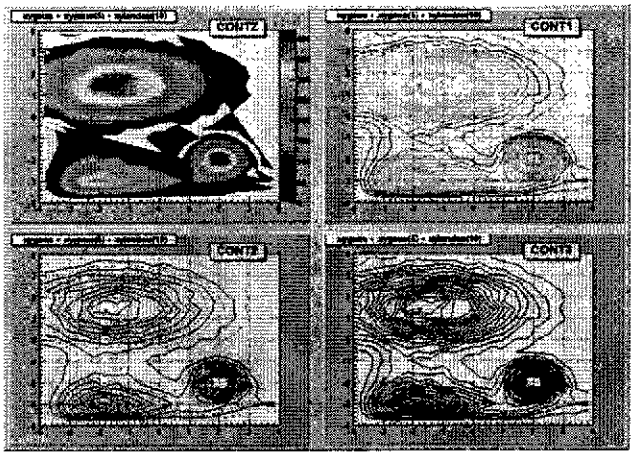


Tout objet dans un canvas peut être édité à la souris

V.6.2 Options 2D

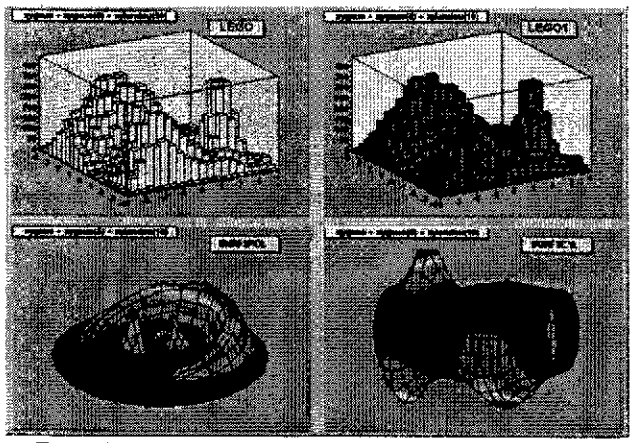


On peut avoir plusieurs vues du même objet

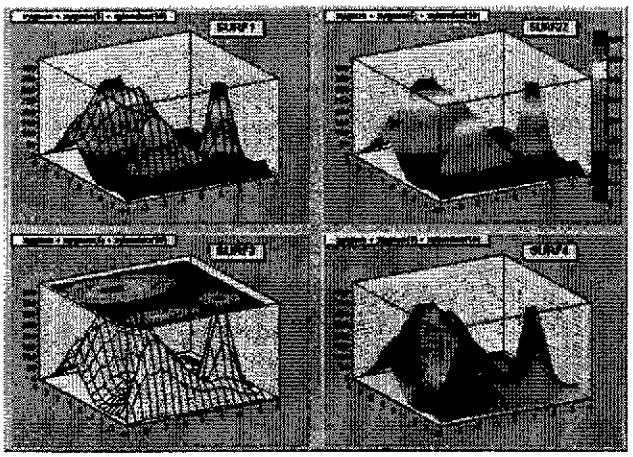


Le nombre et le type des contours peut être modifié avec la souris. Chaque contour est un objet

V.6.3 Options 3D



Rotation et zoom interactifs sont disponibles



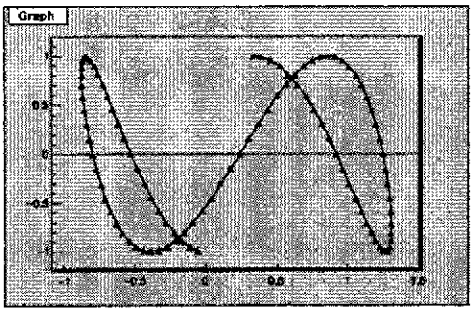
L'image sur l'écran est la même que celle obtenue sur la sortie Postscript

V.7 Outils "physiques"

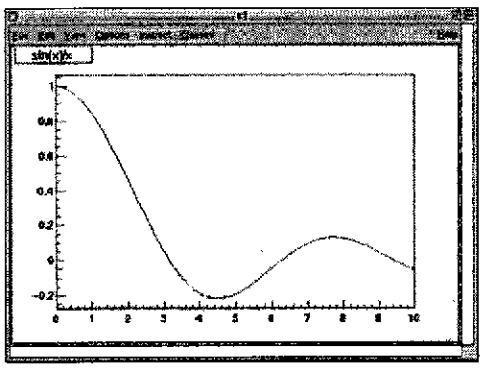
Par "outils physiques", nous entendons les outils de base nécessaires à l'analyse des données recueillies en physique corpusculaire. Nous ne chercherons qu'à survoler les plus importants. Parmi eux, les graphes, fonctions, histogrammes 1D, 2D, 3D et les outils de minimisation seront décrits.

V.7.1 Graphes

Un ensemble de points (x,y) peut être représenté par un graphe (classe TGraph). Les options graphiques supportées sont celles correspondant au cas 1D.



V.7.2 Fonctions



Les fonctions 1D, 2D et 3D (classes TF1,2,3) sont prévues. Un objet TF1 est une fonction à 1 dimension définie entre un minimum et un maximum. La fonction peut être une fonction mathématique simple, définie par une formule ou une fonction, au sens programmation, pré-compilée fournie par l'utilisateur, ou encore une fonction dans un script.
Dans le premier cas, les fonctions mathématiques standard (fonctions transcendantales) de la librairie mathématique C sont reconnues, ainsi que les fonctions logiques.
Dans les deux derniers cas, la fonction (programme) renvoie la valeur de la fonction (mathématique) en un point donné.
Dans tous les cas, la fonction peut également avoir des paramètres associés. Un exemple de création et d'affichage d'une fonction :

```
root > TF1 f1("f1","sin(x)/x",0,10);
root > f->Draw();
```

Essayez !

V.7.3 Histogrammes

Les histogrammes sont sans doute l"un des outils les plus utilisés par les physiciens dans notre spécialité. Dans ROOT, il existe des classes d'histogrammes à 1, 2 ou 3 dimensions. Pour mémoire, PAW ne disposait pas d'histogrammes à 3 dimensions.
Les templates n'étant pas encore utilisés de façon extensive, pour des raisons dont nous n'avons pas le loisir de discuter ici, chaque classe est divisée en un certain nombre de sous classes correspondant au type des données enregistrées dans les histogrammes. Ainsi la classe dénommée TH2F représente les histogrammes à deux dimensions dont le contenu de chaque canal est enregistré sous forme d'un "float" (nombre réel).
La nom des classes est THXY où X représente la dimension, 1 , 2 ou 3, et Y le type de la donnée, C pour "char", S pour "short", F pour "float", D pour "double". Pour être pratique, nous allons illustrer l'utilisation de ces classes.

Sauvegarde/Lecture d'un histogramme de/vers un fichier ROOT

Les lignes suivantes créent un fichier ROOT et y sauvent un histogramme rempli préalablement de données ayant une distribution gaussienne. Nous les avons déjà vues au début du chapitre sur ROOT.

```
root > TFile f("histo.root","new");
```

```
root > TH1F h1("hgaus","histo gaussienne",100,-3,3);
root > h1.FillRandom("gaus",10000);
root > h1.Write();
```

Pour lire cet histogramme dans une autre session ROOT, il faut effectuer :

```
root > TFile f("histos.root");
root > TH1F *h = (TH1F*)f.Get("hgaus");
```

Une fois un fichier ouvert en lecture (ici, variable "f"), la fonction TFile::Get() charge un objet nommé en mémoire et renvoie un pointeur sur cet objet. Ce pointeur est de type TObject, donc on le transforme dans le bon type avant de l'utiliser. Lorsqu'on a un grand nombre de fichiers, on peut sauvegarder TOUS les histogrammes en mémoire dans le fichier par la seule commande:

```
root > f.Write();
```

Remplissage et accès aux données des histogrammes

Un histogramme est rempli, généralement dans une boucle, à l'aide de commandes telles que :

```
h1->Fill(x);
h1->Fill(x,w); //remplissage avec poids
h2->Fill(x,y);
h2->Fill(x,y,w);
h3->Fill(x,y,z);
h3->Fill(x,y,z,w);
```

Si TH1::Sumw2() a été appelé avant le remplissage, la somme des carrés des poids sera également sauvegardée. Il est possible d'incrémenter directement un canal particulier via TH1::AddBinContent() ou remplacer le contenu existant par TH1::SetBinContent().
L'accès au contenu d'un canal (bin) donné se fait par :

```
Double_t bincontent = h->GetBinContent(bin);
```

"Binning" et "rebinning" automatique

Par défaut, l'utilisateur définit les limites et nombre de canaux de l'histogramme à la création de celui-ci et ils restent constants au fur et à mesure du remplissage. Lorsque l'option de "binning" automatique est sélectionnée, la fonction Fill va étendre les limites des axes pour les adapter à la valeur passée en argument. Si on passe un argument plus grand que la valeur maximale, ou plus petit que la valeur minimale, celle ci sera ajustée.
Ce binning automatique est utilisé dans TTree::Draw pour histogrammer les variables d'un arbre sans connaître l'étendue de leurs valeurs.
Un histogramme peut également être "rebinné" (changement du nombre de canaux) à l'aide de TH1::Rebin(). Les incertitudes associées à chaque canal sont recalculées durant ce processus.

Incertitudes associées

Au contenu de chaque canal est associée une incertitude. Le calcul de celle-ci dépend du fait que la somme des carrés des poids de chaque canal soit enregistrée ou non. Par défaut, pour chaque canal, on calcule la somme des poids au remplissage. On peut également appeler TH1::Sumw2 pour forcer la sauvegarde de la somme des carrés des poids pour chaque canal.
Si Sumw2 a été appelé, l'incertitude par canal (bin) est calculée comme racine(somme des carrés des poids), sinon elle est mise à racine(contenu du canal).
Pour obtenir l'incertitude correspondant à un canal donné, faire :

```
Double_t err = h->GetBinError(bin);
```

Où "bin" est le numéro du canal recherché.

Opérations sur les histogrammes

Un grand nombre d'opérations sont implémentées sur les histogrammes. Entre autres :

- Ajout, produit, quotient d'un histogramme par celui qui est courant
- Ajout, produit, quotient de deux histogrammes avec des coefficients et sauvegarde dans l'histogramme courant
- Ajout, produit, quotient d'un histogramme par une fonction

Les incertitudes sont recalculées en supposant des histogrammes indépendants. Les opérateurs +, -, *, / sont supportés.

Projections

Lorsqu'on a un histogramme à deux ou trois dimensions, il est courant de vouloir projeter les valeurs sur l'un des axes. Sont prévues dans ce contexte la projection 1D d'un histogramme 2D ou Profile. Nous laissons au lecteur le soin de regarder la documentation des fonctions TH2::ProjectionX,Y, TH2::ProfileX,Y, TProfile::ProjectionX

Il existe également des projections 1D, 2D ou un profile d'un histogramme 3D. Voir TH3::ProjectionZ, TH3::Project3D.

Toutes ces projections peuvent être ajustées par l'utilisation de TH2::FitSlicesX,Y, TH3::FitSlicesZ. Nous allons reparler de façon plus générale de l'ajustement dans un moment.

Fonctions associées

Un ou plusieurs objets, typiquement des fonctions TF1*, peuvent être ajoutés à la liste de fonctions associées à un histogramme. Soit un histogramme h, on peut récupérer une fonction associée par :

```
TF1* myfunc = h->GetFunction("myfunc");
```

Dessin et affichage des histogrammes

Lorsque vous appelez la méthode Draw (TH1::Draw()) d'un histogramme pour la première fois, un objet THistPainter est créé et son pointeur sauvé comme donnée membre de l'objet histogramme. La classe THistPainter est spécialisée dans le dessin des histogrammes. Elle a été séparée de l'histogramme lui-même pour permettre l'utilisation des objets histogrammes sans la surcharge due au graphique, dans un programme batch par exemple.

Lorsqu'un histogramme est modifié, pas besoin d'appeler Draw() de nouveau. Son image est rafraîchie lorsque le Pad est rafraîchi. Rappelez vous que ce que l'on voit est le reflet de ce qui existe en mémoire. De même, lorsqu'on détruit un histogramme, son image est automatiquement enlevée du pad.

Les éventuelles fonctions associées à l'histogramme sont tracées automatiquement en même temps que lui.

On peut dessiner le même histogramme dans des pads différents avec des options graphiques différentes.

V.7.4 Nombres aléatoires et histogrammes

En physique corpusculaire, la simulation des événements nécessite la génération d'un grand nombre de données aléatoires. Il est donc impérieux que cette génération soit réalisée correctement. Plusieurs générateurs de nombres aléatoires sont prévus dans ROOT. La documentation se trouve dans les classes TRandom, TRandom2 et TRandom3.

On peut remplir aléatoirement un histogramme par TH1::FillRandom selon une distribution donnée. Cette distribution suit soit une fonction analytique de type TF1 existante, soit un autre histogramme, ceci pour toutes les dimensions.

Donnons un exemple. Les deux lignes suivantes créent et remplissent un histogramme de 10000 entrées correspondant à une distribution gaussienne (moyenne = 0, sigma = 1 par défaut) :

```
TH1F h1("h1","histo gaussienne",100,-3,3);
```

```
h1.FillRandom("gaus",10000);
```

TH1::GetRandom est utilisé pour renvoyer un nombre aléatoire distribué selon le contenu d'un histogramme.

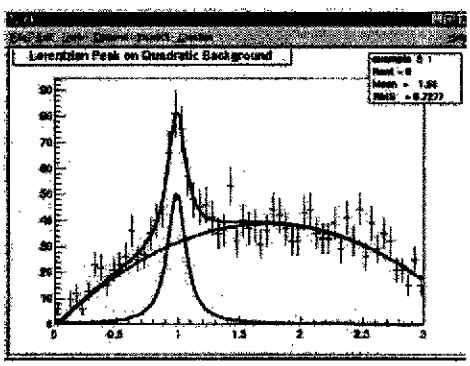
V.7.5 Ajustement ("fit") avec Minuit

Lorsqu'on a une série de valeurs, il est commun de vouloir ajuster une courbe théorique à celles-ci. La librairie d'ajustement Minuit, utilisée depuis de longues années dans PAW a été adaptée et intégrée dans ROOT et c'est l'outil principal d'ajustement de courbes.
On peut utiliser directement Minuit ou bien se laisser porter par l'interface, dans le cas des classes d'histogrammes. La première chose est de définir une fonction que l'on désire ajuster. Ensuite, les histogrammes (1D, 2D et 3D et Profiles) peuvent être ajustés avec cette fonction utilisateur via TH1::Fit. Deux algorithmes d'ajustement sont possibles, la méthode du χ^2 et celle du maximum de vraisemblance (Log Likelihood).
Les fonctions utilisateur peuvent être soit des fonctions standard

- "gaus" pour une gaussienne,
- "landau" pour une distribution de landau,
- "expo" pour une exponentielle,
- "poln" pour un polynôme de degré n.

ou une combinaison de ces fonctions, par exemple "poln+gaus".
Il peut s'agir aussi d'une fonction C++ interprétée ou compilée.

Lorsque l'ajustement d'un histogramme a été fait, la fonction résultante et ses paramètres est ajoutée à la liste de fonctions que possède chaque histogramme. Si l'histogramme est sauvegardé, cette liste également. De plus, la fonction est tracée automatiquement si l'histogramme a déjà été affiché.



On peut récupérer les paramètres de la fonction/ajustement à l'aide de commandes telles que :

```
Double_t chi2 = myfunc->GetChisquare();
Double_t par0 = myfunc->GetParameter(0); // valeur 1er parametre
Double_t err0 = myfunc->GetParError(0); // erreur 1er parametre
```

Combinaison de fonctions

Il est possible d'ajuster une combinaison de fonctions, par exemple un signal, au hasard un pic ayant pour forme une lorentzienne, et un bruit de fond.
$y(E) = a_1 + a_2E + a_3E^2 + A_P (\Gamma / 2\pi)/((E-\mu)^2 + (\Gamma/2)^2)$ bruit de fond pic = lorentzienne

Les paramètres pour le bruit de fond et le signal sont à priori dans 2 tableaux séparés

par[0] = a_1	par[0] = A_P
par[1] = a_2	par[1] = Γ
par[2] = a_3	par[2] = μ

Pour utiliser cette combinaison de fonctions, on met tous les paramètres dans un seul tableau et la fonction à ajuster devient :
FonctionFinale = fond (x, par) + signal (x, &par[3])
avec

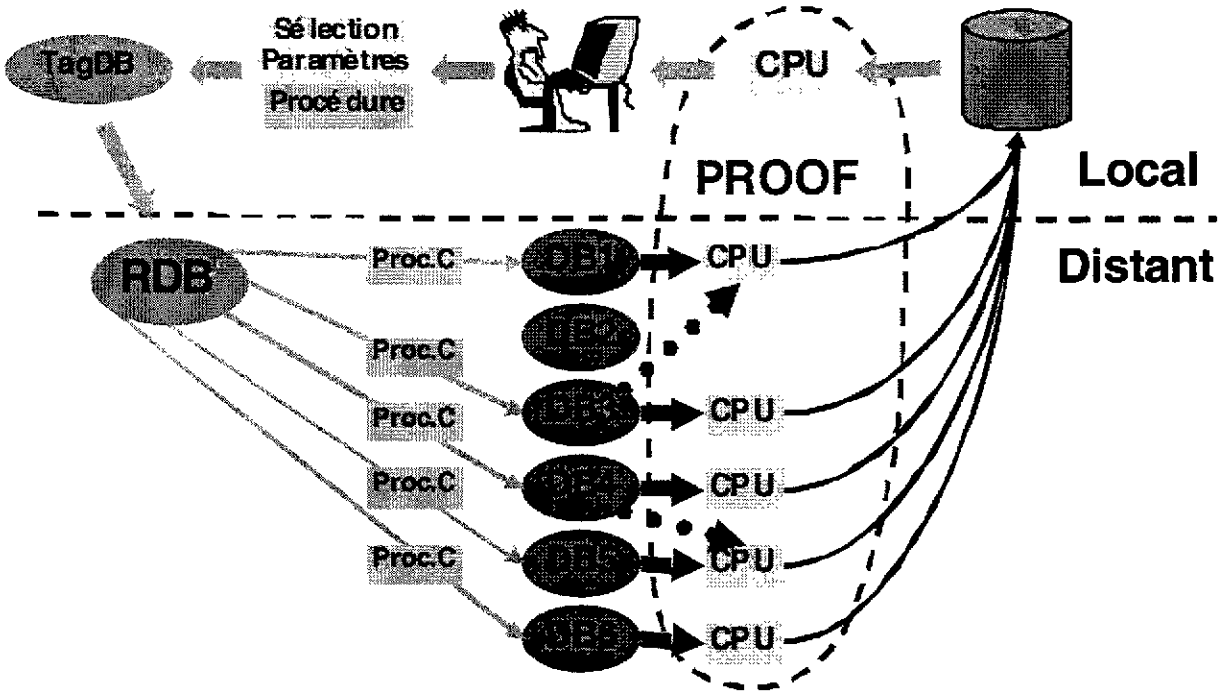
par[0] = a_1
par[1] = a_2
par[2] = a_3
par[3] = A_p
par[4] = Γ
par[5] = μ

V.8 Outils "techniques"

Afin de répondre à des besoins élémentaires, certains outils que nous appellerons "techniques" ont été développés et inclus dans ROOT. Ce sont principalement des conteneurs, c'est-à-dire des objets destinés à contenir d'autres objets. On pense par exemple aux listes, cartes, tableaux.
Ces fonctionnalités ont été développées à une époque où les fonctions équivalentes n'existaient pas ou peu au niveau du langage C++. Aujourd'hui, C++ intègre l'équivalent sous la forme de la librairie STL.
Ceci dit, certains conteneurs sont particulièrement efficaces dans le cadre ROOT, nous pensons en particulier aux TClonesArray.
Le contenu d'un de ces conteneurs est parcouru à l'aide d'un "itérateur" qui renvoie à chaque appel l'objet suivant.
Ici également, nous renvoyons le lecteur à la documentation ROOT pour avoir plus de précisions.

V.9 Calcul parallèle

Nous avons parlé de la nécessité de prévoir la possibilité de faire du calcul en parallèle sur des fermes de machines ou bien sur des sites distribués. PROOF est une implémentation d'un système de calcul parallèle dans ROOT, en cours d développement à l'heure ou sont écrites ces lignes. Il permet le traitement en parallèle de chaînes d'arbres sur des grappes de machines hétérogènes.
Ce développement se fait avec en arrière plan les développements récents de GRID, ce qui nous permet d'en dire un mot. GRID est un concept d'un ensemble de machines dispersées sur la terre devant traiter une très grande quantité de données en parallèle. L'exemple simpliste est le projet SETI@home qui recherche des messages venant d'une éventuelle civilisation extraterrestre dans le signal capté par le radiotéléscope d'Arecibo aux Etats-Unis. Ce signal est découpé en petites tranches, chaque petite tranche est ensuite envoyée à un ordinateur connecté au réseau dont le propriétaire a accepté de faire un peu de travail. Ceci est bien sur automatique, le résultat de la recherche étant renvoyé au site américain. Ainsi, plusieurs millions de machines passent une partie de leur temps à traiter les données du radiotéléscope.
Nous avons dit que ce projet était simpliste parceque la quantité de données reçue dans chaque paquet est très faible, quelques kilooctets. En ce qui nous concerne, ce sont souvent des mégaoctets qu'il faut transférer à chaque événement, et il est hors de question de faire ceci sur une ligne téléphonique, par modem.
Plusieurs possibilités s'offrent à priori pour structurer un système de calcul distribué. Il faut tenir compte de la puissance d calcul disponible sur chaque machine, mais également de la granularité des échanges, c'est-à-dire de la quantité de données échangées à chaque transaction, ainsi que de la capacité des lignes de connexion entre les machines.
Mais on se rend rapidement compte que certaines options ne sont pas adaptées, généralement parcequ'elles induisent un trop grand transfert de données.
Dans le cadre de ROOT et des outils GRID en développement de part le mond, un système de calcul parallèle est en cours de mise au point, la "Parallel ROOT Facility", ou PROOF.



La figure ci-dessus montre la philosophie adoptée. Les serveurs esclaves DBxx forment la partie active, demandant au serveur maître du travail lorsqu'ils sont prêts. Celui-ci envoie des paquets de données à chaque serveur esclave. Les paramètres cruciaux sont la taille des paquets envoyés et la localisation des données. Le maître doit faire attention de n'envoyer à l'esclave que ce qu'il peut traiter, ni trop, ni trop peu. Ceci se décide en fonction du comportement précédent de l'esclave, sa vitesse, un crash éventuel, etc... Au maître de faire également attention à envoyer du travail sur des données qui sont proches de l'esclave, si possible sur l'esclave lui-même, pour minimiser l'occupation du réseau.
Autant que possible, on essaie de transférer la tâche vers les données et non l'inverse. En effet, une tâche est souvent un programme qui ne sera transféré qu'une seule fois, alors que le nombre d'événements à transférer est très grand.
Une fois que chaque processeur a fait son travail, il renvoie un résultat partiel au maître, résultat qui représente un bien plus petit flux de données que ce que le flux entrant. Les résultats partiels sont recombinés par le maître.

Exemple d'une session PROOF

Une session PROOF complète est montrée ci-après.

```
root [0] .! ls -l run846_tree.root
-rw-r--r--   1 rdm    cr   593223259 Feb 1  15:20 run846_tree.root

root [1] TFile f("run846_tree.root")

root [2] gROOT->Time()

root [3] T49->Draw("fPx")
Real time 0:0.11, CP time 10.860

root [4] gROOT->Proof()
*** Proof slave server : pcna49a.cern.ch started ***
*** Proof slave server : pcna49b.cern.ch started ***
*** Proof slave server : pcna49c.cern.ch started ***
```

```
*** Proof slave server : pcna49d.cern.ch started ***
*** Proof slave server : pcna49e.cern.ch started ***
Real time 0:0:4, CP time 0.140

root [5] T49->Draw("fPx")
Real time 0:0:3, CP time 0.240
```

On ouvre un fichier ROOT de 600 Mo contenant un arbre. Si l'on histogramme la variable fPx contenue dans cet arbre, il faut 11 secondes de temps de traitement. Si l'on lance une session PROOF et que l'on redemande le même calcul, chaque serveur esclave va recevoir ses paquets de données et renvoyer un résultat d'histogrammation partiel. Les résultats seront ensuite combinés par le maître pour affichage. Le temps total nécessaire dans ce cas passe à moins de 3 secondes avec 5 machines esclaves. Ceci est bien sûr un cas d'école et les résultats peuvent varier selon l'algorithme. Ceci dit, les données de physique corpusculaire se prêtent bien à ce genre de parallèlisation.

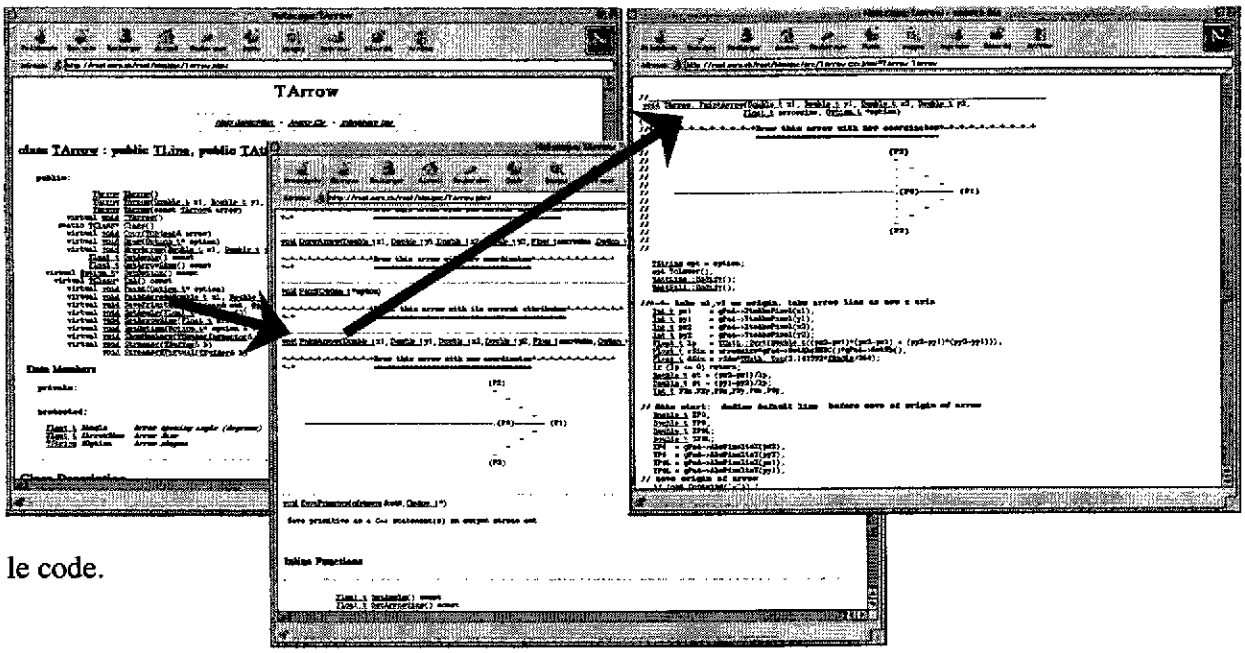
V.10 Documentation automatique

La documentation est très importante dans tous les grands projets. Il faut imaginer que le code source va être produit par plusieurs dizaines voire centaines de personnes. On imagine bien ce qui se passerait si un programmeur passait son code non documenté à un autre. Celui-ci serait tenté de tout refaire !

Mais documenter du code est une tâche ardue, qui demande de la discipline de la part de chacun. En quoi consiste exactement une documentation de code ? Pour l'utilisateur, il s'agit de disposer de manuels, d'exemples, etc... Pour le développeur, il faut une documentation du code, de la structure, etc...

Autant simplifier la tâche des utilisateurs, ou plutôt développeurs en réalisant une documentation automatique à partir du code source.

ROOT dispose d'un mécanisme de documentation automatique basé sur le dictionnaire généré par l'interpréteur CINT à partir des en-têtes de classe, les fichiers "xxx.h" (voir la section sur l'interpréteur). Ce mécanisme produit des pages web, avec des liens permettant de naviguer dans



le code.

Allez voir vous-même ! L'adresse web de la documentation du code source de ROOT est ***http://root.cern.ch/root/htmldoc/ClassIndex.html***

V.11 Et ce n'est pas tout...

Nous ne vous avons pas tout dit, loin s'en faut... Pour plus de détails, vous pouvez consulter ;

- Le site web : ***http://root.cern.ch***
- Le manuel ROOT

Sur le web vous trouverez, en plus de la documentation, les sources et binaires pour une trentaine de combinaisons compilateurs/plateformes.

Bibliographie

1. M. Asai, Object Oriented Design and Implementation, CERN School of Computing, Marathon, Greece, 2000.
2. T. Lewis, Object Oriented Application Frameworks, Manning Publications co., 1995, ISBN 0132139847.
3. R.G. Jacobsen, Storage and Software Systems for Data Analysis, CERN School of Computing, Marathon, Greece, 2000.
4. C. Delannoy, Programmer en langage C++, Editions Eyrolles, 1998, ISBN 2-212-09019-6.
5. Site web officiel Java, http://java.sun.com
6. T. Leflour, LAPP, communication privée
7. M. Asai, Geant4, CERN School of Computing, Marathon, Greece, 2000.
8. J. Apostolakis et al., Geant4 Status and Results, Proceedings of the CHEP2000 Conference.
9. J.Apostolakis, Geant4 status and applications, CERN Computing Seminar, 12 Dec. 2001.
10. Site web Geant4, http://geant4.web.cern.ch/geant4
11. Site web Anaphe, http://anaphe.web.cern.ch/anaphe
12. Site web Jas, http://jas.freehep.org
13. Site web Open Scientist, http://www.lal.in2p3.fr/OpenScientist
14. Rene Brun and Fons Rademakers, ROOT - An Object Oriented Data Analysis Framework, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86.
15. C++ Interpreter - CINT, Masaharu Goto, CQ publishing, ISBN4-789-3085-3 (Japonais)
16. Site web ROOT, http://root.cern.ch